Completeness in SMT-based BMC for Software Programs

Malay K. Ganai
NEC Labs America, Princeton, NJ 08540
malay@nec-labs.com

Aarti Gupta
NEC Labs America, Princeton, NJ 08540
agputa@nec-labs.com

Abstract

Bounded Model Checking (BMC) is incomplete without a completeness threshold ($\mathcal{CT}$) bound. Previous methods, using recurrence diameter for obtaining $\mathcal{CT}$, check for existence of a longest loop-free path at every depth k. For terminating software programs, we propose an efficient method for obtaining $\mathcal{CT}$ that requires solving a formula of size $O(k)$ at some depths only, as compared to previous methods that require solving a formula of $O(k^2)$ (or $O(klogk)$) size at every depth. We augment previous methods for BMC simplifications using model transformation and control flow information, with context-sensitive analysis. This results in more BMC simplifications and further reduction in the number of $\mathcal{CT}$ checks. We have implemented our techniques in a Satisfiability Modulo Theory (SMT)-based BMC framework. Our controlled experiments on real-world software programs show that our proposed formulation provides significant improvements over previous approaches.

1. Introduction

Bounded Model Checking (BMC) [3] has been successively applied to verify real-world designs. BMC is a model checking technique where the falsification of a given Linear Temporal Logic (LTL) property ϕ is checked at a given sequential depth. Typically, it consists of the following steps: unrolling of the design for k time frames, translating the BMC instance into a decision problem ψ such that ψ is satisfiable *iff* ϕ has a counter-example of depth (less than or) equal to k, and using a decision procedure to check if ψ is satisfiable. In Satisfiability Modulo Theory (SMT)-based BMC, ψ is a quantifier-free formula (QFP) in a decidable subset of first order logic, which is then checked for satisfiability by an SMT solver. With the growing use of high-level design abstraction to capture today's complex design features, the focus of verification techniques has been shifting towards using SMT solvers [5, 14, 8], and SMT-based BMC [2, 9], which can potentially provide more scalable alternatives than SAT-based [10] or BDD-based methods [13]. BMC, in general, is incomplete unless checking is performed up to the completeness threshold ($\mathcal{CT}$) bound [3, 15, 4].

Problem Statement: In general, computing $\mathcal{CT}$ bound is computationally expensive. In a typical verification scenario with multiple properties to resolve, it is not often clear how to devise a good verification procedure, i.e., how to balance the limited time resource between proving and falsifying the correctness properties. Therefore, it is important to reduce the time for computing completeness threshold.

1.1. Related Work

For a safety property $\mathbf{G}p$ (where p is a non-temporal expression), optimum $\mathcal{CT}$ is shown [3] to be equal to the *reachability diameter* rd, i.e., the longest shortest path from the initial state. Finding rd requires solving a Quantified-Boolean Formula (QBF) with increasing k, and is computationally expensive. Instead, one can compute the *recurrence reachability diameter* rrd, i.e., the longest loop-free path, by computing a series of SAT checks with increasing k [3]. Such computation requires solving SAT instances of size $O(k^2)$. Thus, each $\mathcal{CT}$ check for computing rrd is quadratic in size of depth and becomes harder to solve with unrolling. In a related work [12], the size of the formula can be further reduced to $O(k\ logk)$ using a *sorting network*. In practice, however, the approach has a limitation since the optimal size of a sorting network for an arbitrary input size is unknown. As every shortest path is a loop-free path, rrd over-approximates rd, i.e., $rrd \geq rd$ and hence, $\mathcal{CT}$ so obtained is sub-optimal.

Although BMC has been used primarily for verifying hardware designs, it has also been used in verifying low-level software programs in C [6, 11]. Most application software programs terminate. Embedded software programs, which are typically reactive, do not terminate; however, parts of the software such as loops must terminate for correct functionality [7]. Typically, for embedded programs that require a high degree of reliability, dynamic memory allocations and recursion are discouraged.

1.2. Our Approach: Overview

We focus on devising efficient proof techniques for terminating software programs in an SMT-based BMC framework. Specifically, we focus on computing rrd efficiently

978-3-9810801-3-1/DATE08 © 2008 EDAA

for determining completeness threshold for verifying low-level embedded programs under the assumptions of finite recursion and finite data. We formulate common design errors such as array bounds violations, null pointer dereferences, use of uninitialized variables, and user-provided assertions as reachability properties, and solve them using BMC. Specifically, our contributions are:

1. We propose a new formulation for determining $\mathcal{CT}$ that requires solving an SMT/SAT formula of size $O(k)$ corresponding to the longest non-terminating path (NTP) in the program. We show that for a terminating program, the length of the longest NTP corresponds to the recurrence diameter of the corresponding extended finite state machine (EFSM). Using control flow information, our formulation eliminates the need to check for $\mathcal{CT}$ at every BMC unroll depth (Sections 3-4).
2. We augment previous methods [9] for BMC simplifications using model transformation and control flow information, with context-sensitive analysis. This allows the previous methods to be applicable for a model with an *irreducible* control flow graph (CFG), which results in more BMC simplifications and further reduction in the number of $\mathcal{CT}$ checks. Frequently, the CFG become irreducible due to function calls not being inlined (Section 5).
3. By reducing the time for computing completeness threshold, we provide a good balance between the falsification and proof methods, thereby, obtaining an effective verification procedure. Our controlled experiments on real-world software programs show that our proposed techniques result in several orders-of-magnitude improvement in performance, compared to previous approaches in SMT/SAT-based BMC (Section 6).

2. Preliminaries

2.1. EFSM, Completeness Threshold

An EFSM model M is a 5-tuple (s_0, C, I, D, T) where s_0 is an initial state, C is a set of control states (or blocks), I is a set of inputs, D is an n dimensional space $D_1 \times \ldots \times D_n$ (each point in D denotes a valuation of n datapath variables with possibly infinite ranges), and T is a set of 4-tuple (c, x, c', x') transitions where $c, c' \in C$, and $x, x' \in D$. An ordered pair $< c, x > \in C \times D$ is called a *configuration* or *state* of M. Let $g : D \times I \mapsto B = \{0, 1\}$ denote a Boolean-valued enabling condition (or *guard*), and $u : D \times I \mapsto D$ denote an update function. A transition from a state $< c, x >$ to $< c', x' >$ under enabling predicate $g(x, i)$, and update relation $u(x, i, x')$ is denoted as $< c, x > \xrightarrow{g/u} < c', x' >$. An NOP state is a control state with no update transition, and a single incoming (outgoing) transition. A SINK (SOURCE) state is a unique control state with no outgoing (incoming) transition.

For an EFSM M and an LTL property ϕ, if ϕ holds in M up to k transitions (or depth), we write $M \models_k \phi$, and if ϕ holds in M for all k, we simply write $M \models \phi$. Let $s_i \equiv < c_i, x_i >$ denote a state, and $T(s_i, s_{i+1})$ denote a state transition relation. We define a *path* as a sequence of successive states, i.e.,

$$path(s_{0..k}) \stackrel{def}{=} \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \quad (1)$$

A path has length k if it makes k transitions. $s_{0...k}$ is shorthand for a sequence of states $(s_0, \ldots, s_k)$. A *loop free path* (LFP) is a path where all states in the path are distinct, i.e.,

$$LFP(s_{0..k}) \stackrel{def}{=} path(s_{0..k}) \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j \quad (2)$$

The recurrence diameter, denoted *rrd* (using the definition as in [3, 12]), is the longest LFP in M, i.e.,

$$rrd(M) \stackrel{def}{=} max\{i | \exists s_0 \ldots s_i \wedge LFP(s_{0...i})\} \quad (3)$$

A *completeness threshold* $\mathcal{CT}(M, \phi)$ is defined as the minimum number of cycles such that if ϕ holds up to $\mathcal{CT}(M, \phi)$, it holds in M for all depths k, i.e.,

$$\mathcal{CT}(M, \phi) \stackrel{def}{=} min\{i | M \models_i \phi \rightarrow M \models \phi\} \quad (4)$$

2.2. Building Models from C threads

We briefly discuss our model building step (similar to [11]) from a given C program under the assumption of a bounded heap and a bounded stack. We obtain first a simplified CFG by flattening the structures and arrays into scalar variables of simple types (Boolean, integer, float). We handle pointer accesses using direct memory access on a finite heap model, and apply standard slicing and constant propagation. We do not inline non-recursive procedures to avoid blow up, but bound and inline recursive procedures. From the simplified CFG, we build an EFSM where each block is identified by a unique *id* value, and a control state variable PC denotes the current block *id*. We construct a symbolic transition relation for PC, that represents the guarded transitions between the basic blocks. For each data variable, we construct an update transition relation based on the expressions assigned to the variable in various basic blocks in the CFG. We use *Boolean* expressions and *arithmetic* expressions to represent the update and guarded transition relations. The common design errors mentioned earlier are modeled as ERROR blocks. In this work, we focus on the reachability of such ERROR blocks. In the sequel, an EFSM state is also referred as a program state.

In Fig. 1, we present a sample C program, and its corresponding EFSM M obtained by our modeling. Each box represents a control state (or basic block) and the unique

```
1.  void bar(int x,
2.           int y){
3.    int d;
4.    if (x≥y)
5.      d = x-y;
6.    else
7.      d = y-x;
8.    return d;
9.  }
10. int foo()
11. {
12.   int a=-10,b=5;
13.   a = bar(a,0);
14.   b = bar(b,0);
15.   while(b!=0){
16.    t=bar(a,b);
17.    if (a≤b) a=t;
18.    else b=t;
19.   }
20.   assert(a!=0);
21. }
```

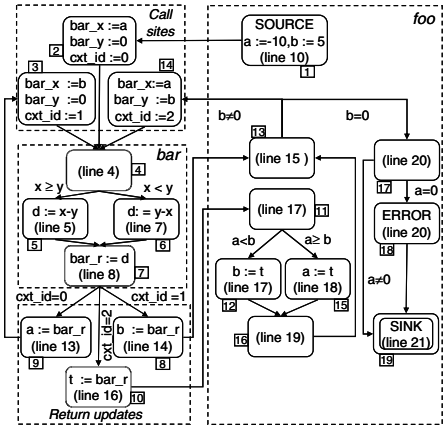


Figure 1. A sample C code and its EFSM M

number in the attached square denotes its *id*. For example, the edge $(4, 5)$ represents a transition from block 4 to 5, predicated on $x \geq y$, with update function $d := x - y$. Blocks 4 and 5 correspond to source lines 4 and 5, respectively. We obtain a CFG by simply ignoring the enabling predicates and update functions. Blocks 4 and 7 are *entry* and *exit* blocks of function `bar`, respectively. Block pairs $(\underline{2}, \underline{9})$, $(\underline{3}, \underline{8})$, and $(\underline{14}, \underline{10})$ correspond to *call* and *return* sites for `bar`. The variable `cxt_id` is introduced to identify the different calling contexts i.e., call/return sites.

2.3. CSR and CFG Transformations

CSR, i.e., control state reachability, is a breadth-first traversal of the CFG (corresponding to an EFSM model), where a control state b is one step reachable from a *iff* there is some enabling transition $a \longrightarrow b$. At a given sequential depth d, let $R(d)$ represent the set of control states that can be reached *statically*, i.e., ignoring the guards, in one step from the states in $R(d-1)$, with $R(0) = c_0$. We say a control state a is CSR*-reachable* at depth k if $a \in R(k)$. For some d, if $R(d-1) \neq R(d) = R(d+1)$, we say the CSR *saturates* at depth d. Computing CSR for the CFG of M (in Fig. 1), we obtain the set $R(d)$ for $0 \leq d \leq 8$ as follows: $R(0) = \{1\}$, $R(1) = \{2\}$, $R(2) = \{4\}$, $R(3) = \{5, 6\}$, $R(4) = \{7\}$, $R(5) = \{8, 9, 10\}$, $R(6) = \{13, 3, 11\}$, $R(7) = \{14, 17, 4, 12, 15\}$, $R(8) = \{4, 18, 19, 5, 6, 16\}$. CSR can be used to reduce the size of BMC instances [9]. Basically, if a control state $r \notin R(d)$, then the unrolled datapath expressions of variables that depend on r can be *simplified* at depth d.

3. Termination-based Completeness Threshold

For terminating programs, we propose a $\mathcal{CT}$ (Eq. 4) that requires solving an SMT/SAT formula of size $O(k)$ at depth k, and show that $\mathcal{CT}$ so obtained corresponds to the recurrence diameter. We define a *non-terminating path* (NTP) as a program path where the last control state c_k (recall, $s_i \equiv < c_i, x_i >$) is not a $SINK$, i.e.,

$$NTP(s_{0\ldots k}) \stackrel{def}{=} path(s_{0\ldots k}) \wedge (c_k \neq SINK) \qquad (5)$$

We define the longest program length (of M), denoted as *lpl*, as the length of the longest NTP, i.e.,

$$lpl \stackrel{def}{=} max\{i | \exists s_0 \ldots s_i \wedge NTP(s_{0\ldots i})\} \qquad (6)$$

Lemma 1 *For a terminating program, each path satisfying $NTP(s_{0\ldots k})$ has distinct states, i.e., $\forall_{0 \leq i < j \leq k}\ s_i \neq s_j$.*

Proof: We prove by contradiction. For some $i < j$, assume $s_i = s_j$. As $NTP(s_{0\ldots k})$ is SAT, $c_i, c_j \neq SINK$. In other words, there exists an NTP path where the program state is revisited. Such a loop would make the program non-terminating, contradicting our assumption. Thus, each state in path satisfying *NTP* has to be distinct. □

Theorem 1 *For a terminating program, $NTP(s_{0\ldots k})$ is SAT if and only if $LFP(s_{0\ldots k+1})$ is SAT.*

Proof: (if:) Given, $LFP(s_{0\ldots k+1})$ is SAT. Thus, $s_k \equiv < c_k, x_k > \neq s_{k+1} \equiv < c_{k+1}, x_{k+1} >$. Since $SINK$ does not have any outgoing transition, clearly $c_k \neq SINK$. Thus, $NTP(s_{0\ldots k})$ is SAT.

(only if:) Given, $NTP(s_{0\ldots k})$ is SAT. Using Lemma 1, $\forall_{0 \leq i < j \leq k}\ s_i \neq s_j$. Further, since $c_k \neq SINK$, there exists an outgoing transition to c_{k+1}. As program terminates, $\forall_{0 \leq i \leq k}\ s_i \neq s_{k+1}$. Thus, $LFP(s_{0\ldots k+1})$ is SAT. □

Clearly from Theorem 1, the recurrence diameter rrd (Eq. 3) corresponds to the longest non-terminating path lpl, i.e., $rrd = lpl + 1$. Comparing formulations of NTP (Eq. 5) and LFP (Eq. 2) at depth k, we observe that the NTP formula has $O(k)$ size, while the LFP formula has $O(k^2)$. In general, to show BMC completeness, one needs to solve an LFP formula for increasing depth k, starting from $k = 0$ [3]. Using the following theorem, we show that with the NTP formulation, together with CSR information, we can skip NTP checks, i.e., satisfiability checks for $NTP(s_{0\ldots k})$ at some k such that $SINK \notin R(k)$. Note, if $SINK \in R(k)$ and $|R(k)| = 1$, we obtain $\mathcal{CT} = k$ immediately.

Theorem 2 *For a terminating program, if $NTP(s_{0\ldots k-1})$ is SAT and $SINK \notin R(k)$, then $NTP(s_{0\ldots k})$ is SAT.*

Proof: As $SINK \notin R(k)$, there is a transition from a c_{k-1} to $c_k \neq SINK$. Clearly, $NTP(s_{0\ldots k})$ is SAT. □

4. SMT-based BMC with NTP check

We present the flow of SMT-based BMC with NTP checks, as shown in Fig. 2. (Shaded blocks 0-2, 5-7 correspond to our contributions.) Note that the flow is applicable to both terminating and non-terminating programs;

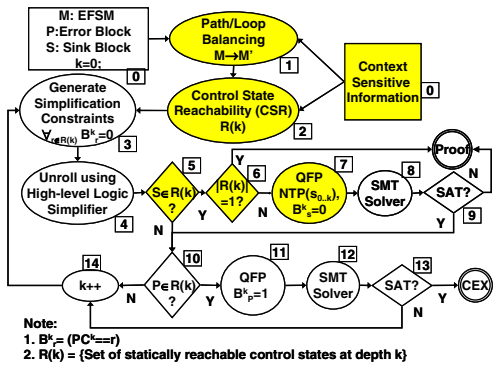


Figure 2. SMT-based BMC (flow)

Step 1: Given *EFSM* M, with an ERROR block P and a SINK block S, perform context-sensitive PB to transform the model to M' using *Algorithm 1* (described in Section 5)
Step 2: Generate context-sensitive CSR information $R(k)$ for the CFG of M' using *Algorithm 2* (described in Section 5)
Steps 3-4: At depth k, use CSR information to simplify the data path expressions that depend on a block that is unreachable at depth k [9]. Use a high-level logic simplifier to simplify the unrolled transition relations.
Step 5: Check if SINK S is CSR-reachable at k. If so, *goto* step 6; else *goto* step 10.
Step 6: Check if S is the *only* control state. If so, *return Proof* with $\mathcal{CT} = k$; else *goto* step 7.
Steps 7-9: Generate a QFP formula for NTP, and check if it is SAT using an SMT solver. If not, return *Proof* with $\mathcal{CT} = k$; else *goto* step 10.
Steps 10-13: Check if block P is CSR-reachable at k. If not, *goto* step 14; else generate QFP formula $B_P^k \equiv (PC^k = P)$ for the error reachability, and check if B_P^k is SAT. If so, *return* counter-example CEX; else *goto* step 14.
Step 14: Increase k, and *goto* to step 3.

Figure 3. SMT-based BMC (algorithm)

however, we will not obtain a $\mathcal{CT}$ bound for the latter. By focusing our proof techniques for terminating programs, we obtain an effective balance between falsification and proof methods. Note, we avoid an expensive LFP check that hardly ever succeeds in practice for non-terminating programs. We describe the flow in Fig. 3, where each step number matches the tagged block in Fig. 2.

Note that calls to an SMT solver for NTP checks are made only when the $SINK$ block is CSR-reachable at that depth. Also, when $R(k) = 0$ for $k > d$ during CSR, we immediately obtain $\mathcal{CT} = d$. In such cases (typically seen in programs without loops), we do not perform NTP checks at all (not shown in the flow). In the following, we discuss techniques that further reduce the number of NTP checks by reducing static reachability of SINK.

5. Context-sensitive PB and CSR

A large cardinality of the set $R(d)$, i.e., $|R(d)|$, reduces the scope of above simplification and hence, the performance of BMC. Re-converging paths of different lengths and different loop *periods* are mainly responsible for the saturation of CSR. Typically, saturation of CSR leads to large $|R(d)|$, and adversely affects the size of the unrolled BMC instances. To avoid saturation, a strategy called *Path/Loop Balancing* (PB) has been proposed [9]. It transforms an EFSM by inserting NOP states such that lengths of re-convergent paths and periods of loops are the same, thereby reducing the statically reachable set of non-NOP control states. Note, an NOP state does not change the transition relation of any variable. The PB techniques [9] are applicable only when the CFG is *reducible*. A reducible graph [1] has the property that there is no jump into the middle of a loop from outside, and there is only one entry node per loop. Note that the CFG of M, shown in Fig. 1, is not reducible, although the corresponding program is *well-structured*, i.e., has only a reducible loop. The introduction of unstructured loops during modeling causes irreducibility. For CFG in Fig. 1, the loop $3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 3$ is unstructured, and is not present in the original program. Such false loops are introduced due to non-inlining of functions. We overcome this problem by making PB strategy context-sensitive as described by *Algorithm 1* in Fig. 4. (One can inline function calls, but EFSM may blow-up in size.)

A run of Algorithm 1: Consider the CFG shown in Fig. 1. The first call to `bar` does not add any NOP states as all paths are balanced. Note, that $wt^{bar} = 2$. When we process `foo`, we add summary edge $(2, 9)$ with weight 4, and remove edges $(2, 4)$ and $(7, 9)$. Similarly, we add summary edges $(3, 8)$ and $(14, 10)$. Then, we apply the PB algorithm [9]. In step 4, we remove the summary edges, and put back the removed edges. Finally, we insert 1 NOP between blocks 17 and 19.

CSR analysis, in general, is not context-sensitive. This leads to large $R(k)$, as many false paths through CFG are considered. We make it context-sensitive as described by *Algorithm 2* in Fig. 5.

A run of Algorithm 2: Consider the CFG shown in Fig. 1. For function `bar`, we obtain $R^{bar}(0)$={4}, $R^{bar}(1)$={5,6}, $R^{bar}(2)$={7}. For function `foo`, we obtain $R^{foo}(0)$={1}, $R^{foo}(1)$={2}, $R^{foo}(2)$={4}, $R^{foo}(3)$={5,6}, $R^{foo}(4)$={7}, $R^{foo}(5)$={9}, and so forth.

To show the effect of context-sensitive analysis on PB and CSR, we experimented with various combinations of strategies on a real-world test case `tcas` (air traffic control and avionic system). Note, `PB` refers to context-sensitive path/balancing technique (as the model is irreducible), and `CXT` refers to context-sensitive CSR. We compare CSR in the following settings: (a) `CSR`: model with no `PB` and

Algorithm 1: Context-sensitive PB

Step 1: Order all functions $\mathcal{F}_1 \ldots \mathcal{F}_m$, s.t. $\mathcal{F}_j$ invokes $\mathcal{F}_i$ only if $i < j$. (Note, as recursive functions are inlined for a bounded number, such an order can easily be obtained.) Identify each function $\mathcal{F}_i$ with its entry fin_i and exit $fout_i$ blocks. *Example* (Fig 1): $\mathcal{F}_1$=`bar`, $\mathcal{F}_2$=`foo`. For `bar`, $fin_1 = 4$ and $fout_1 = 7$; and for `foo` $fin_2 = 1$ and $fout_2 = 19$.
Step 2: For each function $\mathcal{F}_i$ ($1 \leq i < m$), obtain the set of *call/return* sites $\mathcal{X}_i = \{(\underline{xin}, \underline{xout}) \mid (xin, fin_i)$ and $(fout_i, xout)$ are edges in CFG$\}$. *Example* (Fig 1): For $\mathcal{F}_1 =$ `bar`, $\mathcal{X}_1 = \{(\underline{2}, \underline{9}), (\underline{3}, \underline{8}), (\underline{14}, \underline{10})\}$.
Step 3: Foreach $\mathcal{F}_j$ with $1 \leq j \leq m$, i.e., starting from the innermost callee function, do steps 3a and 3b:
Step 3a: For every callee function $\mathcal{F}_i$ with $(\underline{xin}, \underline{xout}) \in \mathcal{X}_i$ respectively, remove the edges (xin, fin_i) and $(fout_i, xout)$, and add a *summary* edge $(xin, xout)$.
Step 3b: Invoke the PB [9] algorithm on the CFG with SOURCE as fin_j and SINK as $fout_j$. The algorithm assigns weights to the edges so that all the paths between fin_j and $fout_j$ have equal weights, i.e., the sum of the weights of edges in each path are equal. The initial weights for the non-summary edges are all set to 1. For an added *summary* edge $(xin, xout)$ corresponding to callee $\mathcal{F}_i$, the initial weight is $wt^{\mathcal{F}_i} + 2$, where $wt^{\mathcal{F}_i}$ is the weight of the path between fin_i and $fout_i$ for callee $\mathcal{F}_i$, computed previously.
Step 4: Remove all the added summary edges, and add the edges that were removed in step 3a. For each edge (a, b) with weight w, add $w-1$ NOP blocks between a and b.

Figure 4. Context-sensitive PB

no `CXT`, (b) `CSR+PB`: model with `PB`, but no `CXT`, and (c) `CSR+PB+CXT`: model with `PB` and `CXT`. We present their reachability graphs in Fig. 6(a)-(c) up to depth D. The width of the graph, proportional to $|R(d)|$ where $0 \leq d \leq D$, indicates the scope of BMC simplification. We observed that using method `CSR`, $SINK$ block appears at *every depth* after saturation; using method `CSR+PB`, it appears *every other depth*; and using method `CSR+PB+CXT`, it appears only *once*. As the method `CSR+PB+CXT` reduces $R(d)$ and static reachability of $SINK$ significantly, it also has the largest potential to improve performance of BMC.

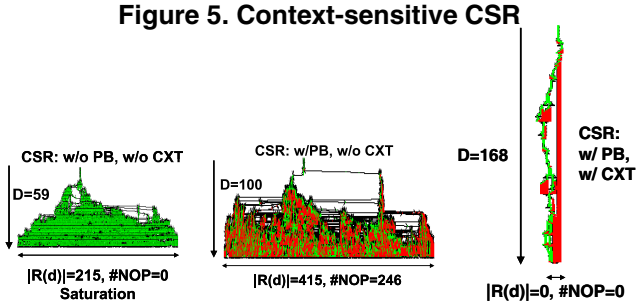
6. Experiments

We have implemented the proposed techniques in our SMT-based verification framework for software programs. We used `yices-1.0` [16], an SMT solver at the backend. We used as benchmarks C programs from public domain and industry, including linux drivers, network application software, and embedded programs in portable devices. Among the 18 benchmarks we considered, `tcas` is an air traffic control and avionic system with 8 *assertions*; `ftpd` is a restart module of *wu-ftpd* with 5 *array bound violation* checks; `mXX` examples are for a network protocol with 110

Algorithm 2: Context-sensitive CSR

Step 1: Order functions s.t. $\mathcal{F}_j$ invokes $\mathcal{F}_i$ only if $i < j$.
Step 2: For each function $\mathcal{F}_i$ ($1 \leq i \leq m$), compute the CSR of the CFG with SOURCE as fin_i and SINK as $fout_i$. We use $R^{F_i}(k)$ to denote the function-specific CSR, i.e., set of control states reachable from fin_i with $R^{F_i}(0) = \{fin_i\}$.
Step 2a: While computing the CSR of the caller $\mathcal{F}_j$, if we reach the entry block fin_i of callee $\mathcal{F}_i$ at depth t, with $[xin, xout] \in \mathcal{X}_i$, we perform a *set-union* operation as follows: $\forall_{0 \leq a < D_i}\ R^{\mathcal{F}_j}(t+a) = R^{\mathcal{F}_j}(t+a) \cup R^{\mathcal{F}_i}(a)$, and $\forall_{1 \leq b < D_i, fout_i \in R^{\mathcal{F}_i}(b)}\ R^{\mathcal{F}_j}(t+b+1) = \{xout\} \cup R^{\mathcal{F}_j}(t+b+1)$. We choose smallest D_i such that $R^{\mathcal{F}_i}(D_i) = \emptyset$. In presence of loops, we may not find such D_i. In that case, we choose some large value.

Figure 5. Context-sensitive CSR



(a) `CSR` (b) `CSR+PB` (c) `CSR+PB+CXT`

Figure 6. Combining CSR with PB/CXT

null pointer de-references checks; and `hYY` examples correspond to software for cell phones with 116 *array bound violation* checks.

Our experiments were conducted on a workstation with 3.4GHz, 2GB of RAM running Linux 2.6.9-1.667. We used a time-out of 1000s for each run. (In practice, verification engineers have to run several examples, and they typically allocate 10-20 minutes for each example.) In order to reduce the overhead of running BMC multiple times for a given example, we run BMC in *multiple check mode*, where all properties (unresolved so far at depth k) are checked at depth k, rather than checking them in separate BMC runs.

We performed a controlled experiment with various strategies, and show the BMC comparison results in Table 1. Column 1 gives the name of the benchmark with number of properties shown in parenthesis. Columns 2-15 provide results of BMC with (+) and without (–) combinations of context-sensitive PB (`PB`), context-sensitive CSR (`CXT`), NTP checks (`NTP`), LFP checks (`LFP`), using solvers `SAT` or `SMT`. Note, `-CXT` denotes CSR without context-sensitive analysis. To illustrate, Columns 2-3 show results for the method `-PB-CXT+LFP+SAT`, i.e., `SAT`-based BMC with `LFP` checks, without *PB* model transformation, and without `CXT`, where Column 2 shows number of proofs (P), witnesses found (W) and unre-

Table 1. Comparing BMC using ±PB, ±CXT, (LFP/NTP), (SMT/SAT)

Ex (#prp)	BMC with(+)/without(–) SMT, SAT, PB: Path/Loop Balance, CXT: ConteXT-sensitive LFP: Loop-Free Path Check, NTP: Non-Terminating Path Check (P≡# Proofs, W≡# Witnesses, TO≡Time-out, D≡BMC Depth)													
	-PB-CXT +LFP+SAT		-PB-CXT +LFP+SMT		+PB-CXT +LFP+SMT		+PB+CXT +LFP+SMT		+PB-CXT +NTP+SMT			+PB+CXT +NTP+SMT		
	P/W/?	D(sec)	P/W/?	D(sec)	P/W/?	D(sec)	P/W/?	D(sec)	P/W/?	D(sec)	#NTP	P/W/?	D(sec)	#NTP
tcas (1)	0/0/1	76(TO)	0/0/1	49(TO)	0/0/1	71(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas0 (1)	0/0/1	82(TO)	0/0/1	22(TO)	0/0/1	73(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas1 (1)	0/0/1	78(TO)	0/0/1	22(TO)	0/0/1	71(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas2 (1)	0/0/1	82(TO)	0/0/1	22(TO)	0/0/1	70(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas3 (1)	0/0/1	76(TO)	0/0/1	22(TO)	0/0/1	70(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas4 (1)	0/0/1	67(TO)	0/0/1	22(TO)	0/0/1	71(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
tcas5 (1)	0/0/1	93(TO)	0/0/1	25(TO)	0/0/1	72(TO)	**1/0/0**	**164**(3s)	0/0/1	87(TO)	5	**1/0/0**	**164**(3s)	0
tcas6 (1)	0/0/1	81(TO)	0/0/1	26(TO)	0/0/1	73(TO)	**0/1/0**	**161**(6s)	0/0/1	86(TO)	5	**0/1/0**	**161**(6s)	0
tcas7 (1)	0/0/1	76(TO)	0/0/1	22(TO)	0/0/1	71(TO)	**1/0/0**	**168**(0s)	0/0/1	83(TO)	3	**1/0/0**	**168**(0s)	0
ng4 (3)	0/2/1	74(TO)	0/2/1	88(TO)	0/2/1	226(TO)	0/2/1	256(TO)	**1/2/0**	**430**(26s)	12	**1/2/0**	**430**(26s)	12
ok4 (2)	0/1/1	67(TO)	0/1/1	94(TO)	0/1/1	233(TO)	0/1/1	257(TO)	**1/1/0**	**331**(17s)	12	**1/1/0**	**331**(17s)	12
hYY1 (12)	0/7/5	61(TO)	0/7/5	41(TO)	0/7/5	92(TO)	0/7/5	97(TO)	0/7/5	137(TO)	13	**0/7/5**	**232**(TO)	9
hYY2 (14)	0/10/4	61(TO)	0/9/5	43(TO)	0/8/6	88(TO)	0/8/6	94(TO)	0/9/5	146(TO)	14	**0/10/4**	**195**(TO)	8
hYY3 (57)	0/7/50	41(TO)	0/9/48	43(TO)	0/11/46	97(TO)	0/11/46	97(TO)	**0/25/32**	**172**(TO)	2	**0/25/32**	**172**(TO)	2
hYY4 (33)	0/9/24	53(TO)	0/9/24	45(TO)	0/11/22	79(TO)	0/14/19	97(TO)	0/15/18	106(TO)	23	**0/21/12**	**184**(TO)	23
mXX1 (22)	0/13/9	115(TO)	0/19/3	79(TO)	0/20/2	165(TO)	0/20/2	184(TO)	**2/20/0**	**288**(132s)	2	**2/20/0**	**288**(129s)	2
mXX2 (88)	0/46/42	30(TO)	0/41/47	29(TO)	0/55/33	81(TO)	0/56/32	90(TO)	**0/56/32**	**118**(TO)	7	**0/56/32**	**118**(TO)	4
ftp (5)	0/2/3	48(TO)	0/2/3	48(TO)	0/2/3	111(TO)	0/2/3	119(TO)	0/3/2	154(TO)	1	**0/3/2**	**261**(TO)	3

solved properties (`P/W/?`), and Column 3 shows number of BMC unrollings (D) performed with runtime (in sec) in parenthesis. As an example, for `mXX1` with 22 properties, `-PB-CXT+LFP+SAT` times out (TO) with 0 proof and 13 witnesses at depth 115. Similar results are presented for other columns using SMT solver. For methods `+PB-CXT+NTP+SMT` and `+PB+CXT+NTP+SMT`, we also present number of NTP checks ($\#NTP$) in Columns 12 and 15, respectively. For methods using `LFP`, *number of LFP checks equals D* (not shown separately), as it is performed at every depth. Note, the time needed for performing `PB` and `CXT` are negligible, and so, we do not report them separately.

We use the strategy `-PB-CXT+LFP+SMT` as our baseline [9] for comparison, as CFGs for the benchmarks were not reducible. Note, for some benchmarks such as $tcas$, using methods `+PB+CXT+LFP+SMT` and `+PB+CXT+NTP+SMT`, we obtain $\mathcal{CT} = 168$ statically, as $R(k) = 0$ for $k > 168$. Thus, for these methods we skip the $\mathcal{CT}$ checks. Note, $tcas$ examples did not have a structured loop, but the models have unstructured loops, which were introduced during the modeling phase. In our controlled experiments, we observe that the techniques `PB`, `CXT` and `NTP` *always* help in resolving more properties, or in performing deeper and faster search, or both. In general, we see far fewer NTP checks compared to LFP checks. Overall, the strategy `+PB+CXT+NTP+SMT` is the clear winner.

7. Conclusion

We described an efficient computation of completeness threshold in SMT-based BMC for terminating software programs. We proposed the use of context-sensitive analysis to overcome the limitation of previous method for BMC simplification. Our approach improves performance of the overall verification procedure as demonstrated by our controlled experiments on real-world benchmarks.

References

[1] A. B. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-wesley Publishing Company, 1988.

[2] A. Armando, J. Mantovani, and L. Platania. Bounded Model Checking of Software Using SMT Solvers Instead of Sat Solvers. In *Proc. of SPIN Workshop*, 2006.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[4] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proc. of FMCAD*, 2000.

[5] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, M. Schulz, and R. Sebastiani. The MathSAT 3 system. In *Proc. of CADE*, 2005.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, 2004.

[7] P. Cousot and R. Cousot. Verification of embedded software: Problems and perspectives. *LNCS*, 2001.

[8] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV*, 2006.

[9] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of ICCAD*, 2006.

[10] M. K. Ganai and A. Gupta. *SAT-based Scalable Formal Verification Solutions*. Springer Science and Business Media, 2007.

[11] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Proc. of CAV*, 2005.

[12] D. Kroening and O. Strichman. Efficient computation of recurrence diamter. In *Proc. of VMCAI*, 2003.

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propogation and its application to difference logic. In *Proc. of CAV*, 2005.

[15] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Proc. of FMCAD*, 2000.

[16] SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.