Bounded Model Checking of Embedded Software in Wireless Cognitive Radio Systems

Nannan He and Michael S. Hsiao (nhe, hsiao}@vt.edu)
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061

Abstract

We present a new verification approach that applies aggressive program slicing and a proof-based abstraction-refinement strategy to enhance the scalability of bounded model checking of embedded software. While many software model-checking tools use program slicing as a separate or optional step, our program slicing is integrated in the model construction and reduction process. And it is combined with the compilation optimization techniques so to compute a more accurate slice. We also explore a proof-based abstraction-refinement strategy using the under/over-approximation on our proposed software model, and propose a heuristic method of deciding new encoding size to refine the under-approximation. Experiments on C programs from wireless cognitive radio systems show this approach can greatly reduce the model size and shorten the solving time by the SAT-solver.

1. Introduction

Recently, software defined and cognitive radios have become increasingly popular in many radio communication services. This is mainly because of the flexibility and adaptability brought by the core software system embedded in radios. At the same time, the reliability of the embedded software is critical to the reliability of these radios. For example, the waveforms defined by the software must be ensured compliant with Federal Communications Commission (FCC) regulations [16]. Any violation may cause band interference that could conceivably affect critical spectrum users and cause disastrous results. Therefore, the rigorous software validation techniques like formal verification are in great need in such applications. With the successful application of model checking for hardware designs, there is a growing interest in applying such technique to formally verifying the embedded software programs written in realistic programming languages like C.

The scalability of model checking [1] is an important issue in validating software since software in general has a much larger state space than hardware. Program slicing and abstraction-refinement are two major techniques to enhance scalability. Program (static) slicing automatically extracts a subset of program segments called slice(s) that involves only the variables referred in the slicing criteria that can be the target property assertion statement. The slice(s) is an accurate abstraction without incurring spurious errors since it includes all the constraints relevant to the computation of the variables referred in the assertion. Abstraction-refinement is a technique that could further enhance the scalability by identifying constraints only relevant to the verification of the assertion. The abstraction starts with a small subset of constraints, and more constraints relevant to the verification are identified and restored in the refinement step. Proof-based [2] and counterexample-guided refinement frameworks [3, 4] are two techniques to automatically learn relevant constraints.

Software model checking has been investigated in [5-10]. Some apply symbolic model checking with predicate abstraction [5, 6, 10] or without abstraction [7]. Others apply SAT-based bounded model checking (BMC) [8-10]. Our approach is in the second category. CBMC [8] is the first SAT-based bounded model checker for embedded software in ANSI-C, to the best of our knowledge. It doesn’t document the scalability enhancement method although the tool supports limited program slicing. Saturn [9] uses BMC to detect hard errors and uses function summaries represented as finite state systems to be scalable to handle inter-procedural calls. F-Soft [10] supports both SAT-based BMC and BDD-based unbounded MC. It uses predicate abstraction and counterexample-guided refinement technique to enhance the scalability.

Since the properties usually depend only on a small portion of the program code our proposed approach aims to accurately and efficiently find this portion. The main contributions of this paper are:

- We integrate an aggressive program slicing approach using compiler optimization techniques into our software model construction and reduction process to significantly reduce the model size.
- We explore a proof-based under/over-approximation abstraction-refinement strategy on our proposed software model for BMC of embedded software. We propose a heuristic method
of deciding updated encoding size to improve refining the under-approximation.

Experimental results on the software programs from wireless cognitive radio systems show the effectiveness of the proposed approach.

The rest of the paper is organized as follows. Our proposed software model reduction is presented in Section 2. We introduce the bounded model checking approach based on proof-based abstraction-refinement strategy in Section 3. Experiments followed by conclusions are given in Sections 4 and 5 respectively.

2. Software model reduction

Our proposed structural software verification model $M$ is built from a Static Single Assignment (SSA) representation of the pre-processed program in which all loops or recursions have been unwound and function calls are inlined. (We omitted the details of model construction.) Model $M$ is a Digraph $\langle V, E \rangle$, in which each node $v \in V$ represents a variable and its computation. Each directed edge $e \in E$ represents a data flow that carries the value computed by its source node to its target nodes. $M$ has two derivations $M_{H}$ and $M_{B}$. The high-level model $M_{H}$ is a data-flow-like representation of the program that could facilitate program slicing. Actually this kind of representation has been widely used for compiler optimization, like Value Dependence Graph (VDG) [11]. The main difference is that $M_{H}$ is built from the pre-processed and SSA-transformed program while VDG is applied to the program with full language features. After the model reduction, the bit-level model derivation $M_{B}$ is built by converting program operators in $M_{H}$ to the corresponding Boolean operators through circuit translation. In $M_{B}$, each node only owns a Boolean operator and each edge carries a vector of Boolean values.

2.1 High-Level model reduction and array modeling via program slicing

We perform static slicing through the major reduction procedures shown in Figure 2. The slicing criterion is set as the target property assertion. Slicing starts from the backward reachability analysis of Basic Blocks (BBs) during model construction. All BBs unreachable to the target assertion are sliced away. But the obtained slice is considered to be coarse because not all computations inside the reachable BBs are necessarily relevant. At this time, the slicing process can go to either procedure 2 or 4 depending on whether the current slice has operations over constants and array variables or not. COI reduction as a separate procedure can be invoked after any procedure 2, 3 and 4. Its purpose is to remove the nodes identified as irrelevant by any invoking procedure. Finally the model reduction stops at procedure 5. Since the COI reduction is widely known, we only introduce the procedure 2, 3, 4.

2.1.1 Constant Propagation

It is a popular compiler optimization technique whose goal is to discover and propagate constants through the program. Any node in $M_{H}$ whose inputs are all constant values can also be evaluated as a constant node and propagated further. We use a simulator similar to a logic simulator to implement the constant propagation on model $M_{H}$. The main difference is the value evaluation can be in the Integer or Floating Point domain instead of being restricted in the Boolean domain. After the propagation, we could remove the redundant constant nodes that have no connections to other non-constant nodes. This removal further reduces the $M_{B}$ size and the logic implication cost from the bits with constant value. In case the constant True (False) is propagated to the condition under which $x_{i}$ is defined. If $g_{i}\#$ is true, $x_{4}$ equals to $x_{i}$; otherwise equals to $x_{2}$. The red node represents the assertion. And the validation problem is converted to checking if its output to be False is satisfiable.

Figure 1. Example SSA and its $M_{H}$

Figure 1 gives an example of a simple SSA-transformed code and its model $M_{H}$. In Figure 1(a), the subscript indices distinguish different variable versions in the SSA. The version $x_{i}$ is defined by a Phi function parameterized with reachable versions $x_{i}$ (i=2,3). In 1(b), $x_{4}$ is modeled as a “c?T:F” node with inputs $g_{i}\#$, $x_{3}$ and $x_{2}$ corresponding to condition $c$, True branch and False branch. $g_{i}\#$ modeled as ($x_{2}$!=1) is the condition under which $x_{3}$ is defined. If $g_{i}\#$ is true, $x_{4}$ equals to $x_{i}$; otherwise equals to $x_{2}$. The red node represents the assertion. And the validation problem is converted to checking if its output to be False is satisfiable.

2.1 High-Level model reduction and array modeling via program slicing

We perform static slicing through the major reduction procedures shown in Figure 2. The slicing criterion is set as the target property assertion. Slicing starts from the backward reachability analysis of Basic Blocks (BBs) during model construction. All BBs unreachable to the target assertion are sliced away. But the obtained slice is considered to be coarse because not all computations inside the reachable BBs are necessarily relevant. At this time, the slicing process can go to either procedure 2 or 4 depending on whether the current slice has operations over constants and array variables or not. COI reduction as a separate procedure can be invoked after any procedure 2, 3 and 4. Its purpose is to remove the nodes identified as irrelevant by any invoking procedure. Finally the model reduction stops at procedure 5. Since the COI reduction is widely known, we only introduce the procedure 2, 3, 4.

Figure 2. Slicing procedures.

2.1.1 Constant Propagation

It is a popular compiler optimization technique whose goal is to discover and propagate constants through the program. Any node in $M_{H}$ whose inputs are all constant values can also be evaluated as a constant node and propagated further. We use a simulator similar to a logic simulator to implement the constant propagation on model $M_{H}$. The main difference is the value evaluation can be in the Integer or Floating Point domain instead of being restricted in the Boolean domain. After the propagation, we could remove the redundant constant nodes that have no connections to other non-constant nodes. This removal further reduces the $M_{B}$ size and the logic implication cost from the bits with constant value. In case the constant True (False) is propagated to the
conditional input \( c \) of the “\( c?T:F \)” node, the connection of the False (True) input branch could be removed as the branch is detected as never taken.

![Figure 3. Array node expansion](image)

2.1.2 Array Modeling and Reduction We first assume all array variables have fixed sizes that is mostly true in our verified programs. We also assume there are only two program operations on array variables. Let \( A \) be an array variable, \( e \) and \( x \) are scalar variables, these two operations are: “\( A[x] = e \)” for implementing \textbf{Store} access and “\( e = A[x] \)” for \textbf{Select} access. Actually other array operations can be transformed to a series of steps involving these two simple operations.

An array node \( A \) is first expanded to a set of nodes each of which represents an element of \( A \). For “\( A[x] = e \)” only the \( x^{th} \) element of \( A \) receives the new value \( e \), while all other elements of \( A \) retain their previous values. Figure 3(a) models the following constraint using the SSA representations of \( A \) imposed by this operation, where \( A_{k-1} \) is the latest version of array variable \( A \) at this operation location:

\[
\forall i \in [0,N-1], A_k[i] = (i==x)? e : A_{k-1}[i].
\]

Figure 3(b) models the following constraints imposed by \textbf{Select} access: 
\[
\land_{i=0..N-1} ((i==x)? (e==A_k[i]) : T).
\]

The modeling in Figure 3 is for the general case that the index \( x \) is not known. After all loop structures are fully unwounded and constant propagation applied, the value \( x \) can almost always be computed. Then the complexity of modeling array access operations can be greatly reduced. For \textbf{Store} accesses, all “\( c?T:F \)” nodes are simply substituted by nodes with “\( = \)” operator as the condition \( c \) is deterministic: the node representing the \( x^{th} \) element \( A_k[x] \) is connected with \( e \) while any other element node \( A_k[i] \) is connected with node \( A_{k-1}[i] \) correspondingly. For \textbf{Select} accesses, only the array element whose index equals to \( x \) is connected with the node representing variable \( e \), leaving the rest array element nodes unconnected from node \( e \).

For the nodes with the equality (“\( = \)”) operator which are redundant in logic reasoning, we perform the transitive equality reduction to slice away these nodes to reduce the model size. For the nodes with no output connection, they can be easily identified and removed during COI reduction. Through this reduction, the model size does not depend on the array size, but on the number of \textbf{Store} and \textbf{Select} accesses on the array. In a word, this reduction exploits the fixed array accesses after the unwinding of loops operating on arrays that the element index referred becomes constant.

2.1.3 Redundant Branches Reduction A motivational example is given in Figure 4. In Figure 4(a), line 2 is redundant to the computation \( y \) in line 10 due to the constant negation between two conditional predicates. This kind of redundancy is reported as undetectable by most traditional slicing techniques without compiler optimization [12]. But it can be identified and removed in following two steps performed on Model \( M_{bf} \):

1. Identify the pairs of conditional predicate nodes that have constant equality or inequality correlation.
2. Determine the redundant computations according to case pattern recognition. Currently we only consider a limited number of redundant case patterns. For one example of Figure 4(b), as two condition predicates are constantly negated and node \( x_d \) is connected with True branch of node \( y_b \), the constraints on True branch of \( x_d \) are identified as redundant shown in blue.

![Figure 4. (a) Code Example. (b) Model \( M_{bf} \).](image)

3. Software bounded model checking

A proof-based abstraction-refinement strategy proposed for bit-vector arithmetic reasoning [13] is explored on the bounded model checking to enhance scalability.

3.1 Major steps in refinement framework

\textbf{Step 1: Under-approximation construction} Every free variable node \( v \) in model \( M_{bf} \) has an encoding size \( S_v \) used for under-approximation. For variables whose original bit-vector width \( T_v \) is greater than \( S_v \), we add constraints on its most significant bits by setting them to a constant value. For the example of unsigned free variables with its most significant bits set to “0”, their value is originally arbitrary in range \([0, 2^{S_v} -1]\) is now restricted in range \([0, 2^{S_v} -1]\). In the implementation, we simply use single-literal clauses to restrict these most significant bits in the CNF (Conjunctive Normal Form) formula. If the solver
returns satisfiable for this constrained formula, the target assertion is disproved with a returned feasible counterexample. This is because the state space of an under-approximation model is a subset of that in the original model so that the counterexample is definitely feasible in the program. If the SAT solver returns unsatisfiable (UNSAT), it is inconclusive. But we could extract the UNSAT core as a proof to guide the following abstraction construction. It is important to keep consistency among the added constraints to avoid UNSAT caused by inconsistent constraints. This is because the SAT solver may return meaningless UNSAT core that is not useful for the following abstraction. So considering the example program equation “y==x+1”. We set the encoding sizes $S_v$ of all node variables that translate this word-level equation in the bit-level model $M_B$ to be the same, and set the most significant bits of these variables to be the same kind of constant constraint: either all “1” or all “0”.

### Algorithm 3.1. Over-approximation

**Step2: Abstraction construction and verification**

Constructing the abstraction model $M_{abs}$ is an over-approximation procedure that extracts a subset of nodes from model $M_B$. The main point of Algorithm 3.1 is to group the set of nodes in $M_B$ according to the translation relationship with the nodes in model $M_H$. So, if any bit of node variables in this set is referred in the UNSAT core, all node variables are included in $M_{abs}$; otherwise they don’t contribute to $M_{abs}$. This is to simplify the abstraction and reduce the non-determinism of the over-approximation. Some popular SAT-solvers provide an UNSAT core if a formula is UNSAT. The core $C$ usually has two parts: a set of bit variables $V$ referred in $C$ and a set of clauses $CL$ involved in $C$. Since all clauses in the formula that use variables in set $V$ are a superset of set $CL$, it is safe to use $V$ to build the abstraction.

**Theorem 1.** $M_{abs}$ is an abstraction of $M_B$.

**Proof.** Since $M_{abs}$ has a subset of nodes in $M_B$, the set of CNF clauses $K_{abs}$ of $M_{abs}$ is also a subset of CNF clauses $K_B$ of $M_B$. Let $\alpha$ be a satisfying assignment of $M_B$. Under this assignment, since all clauses in $K_B$ are true, any subset of clauses must also be true. So $K_{abs}$ must be also satisfied. Conversely, if $K_{abs}$ is UNSAT, then $K_B$ must be UNSAT since all clauses in $K_{abs}$ referred to prove UNSAT are also in set $K_B$.

**Theorem 2.** $M_{abs}$ encoded with $S_v$ is UNSAT.

**Proof.** Since $M_B$ encoded with $S_v$ was UNSAT with Core $C$, and since $M_{abs}$ includes all clauses that refer bit variables in set $V$ of this Core $C$, all clauses that contribute to $C$ are still included after constraining the formula of $M_{abs}$ with encoding size $S_v$.

Based on Theorem 2, all counterexamples within bit-width $S_v$ for variable $v$ can be ruled out from $M_{abs}$. If there is a counterexample in $M_{abs}$, the variable $v$ must have a width larger than $S_v$. Since at least one variable needs to increase its encoding size in every iteration, this refinement procedure is ensured to terminate. If $M_{abs}$ is UNSAT, the assertion proves to be true. Otherwise, a counterexample (potentially spurious) is returned which is used to directly build the under-approximation in next iteration.

**Step3: Computation of encoding size $S_v$ for next under-approximation**

Given a counterexample, each variable in $M_{abs}$ is given a value $O$. One direct way to determine an updated value $S_v$ for each variable in $M_{abs}$ is to let $S_v$ big enough to cover $O$ for the variables not in $M_{abs}$, we derive their $S_v$ by data dependency to keep the consistency among new constraints added by updated encoding size. In a word, we make use of the width of assigned bits in counterexample assignment instead of the assignment itself to direct the next refinement iteration that the counterexample-based refinement does. Note that after this process, different variables may be assigned different updated $S_v$.'
3.2 New encoding size computation

Due to the decision heuristics of the SAT solver, a counterexample may assign large values on variables [14]. So the updated encoding width based on the variables’ values given in the counterexample may be unnecessarily large. Since extracting UNSAT cores may become increasingly difficult with an increasing encoding size, the whole refinement may suffer performance loss. So we propose a heuristic method to improve the computation of the updated encoding size using program analysis. In Algorithm 3.2, when a counterexample $\text{Cex}$ is produced, we first identify any node variable in $\text{MB}$ corresponding to the control predicate in the program and its value $T$ implied in the under-approximation is different from that assigned in $\text{Cex}$. These nodes that decide which program paths to be taken are important to help find the SAT solution in the under-approximation. Then we decide their updated $S'_v$ to allow free values in the next under-approximation. Then, we use the data dependency to determine $S'_v$ of other nodes. For example ($x>100$) with old $S_v=4$ is constantly false, $S'_v$ of $x$ is then set 7 to avoid the constant value in next under-approximation. Since many control predicates have a constant RHS in our verification problem, it is easy to decide $S'_v$.

4. Experimental results

We implemented the proposed approach in C++, which is called C2BIT and used it to verify properties in C programs from a wireless cognitive radio system [16]. Our benchmark programs are extracted from two safety-critical components in this system. One is the policy engine that enforces regulatory restrictions on the waveform. Another is the cognitive controller. Two important features in these programs are:

1. Most loop structures have an upper bound so the execution always terminates. For example there are 10 C programs in the controller with 23 for loops whose upper bound is explicit; 15 while loops, twelve of which the maximum bound is statically known, the rest 3 loops are “while (true)” for monitoring sockets with very simple operations that are not our verification targets.

2. Most Array variables have constant size.

With all loops unrolled to a certain bound, all function calls inlined and properties specified as assertions, we use open source GCC 4.0 compiler to generate the SSA form for the pre-processing. The maximum bound $K$ for unrolling can be statically identified for formally proving the program correctness. Then C2BIT uses this form as input for BMC. During model checking with refinement, C2BIT uses zChaff to check the under-approximation and extract UNSAT core as proof in UNSAT case, and uses MINISAT to check the satisfiability of the over-approximate abstraction. This is because zChaff provides a more user friendly UNSAT core compared with MINISAT. All experiments run on Intel Xeon 2.8GHz processor with 2 GB RAM.

Figure 5. Results on bubble sort

In Figure 5, we first show the effectiveness of our slicing method using a simple BubbleSort code [15] compared to CBMC 2.4 with --slice option. The property we checked is $\text{assert}(A[N-2]<A[N-1])$ where $A$ is array variable, $N$ is $\text{size}(A)$. All loops are maximally unwound. The left plot shows the CNF formula size generated by CBMC grows very fast. While the growth of C2BIT is almost proportional to $N$. The result also confirmed our claim that the CNF formula size generated from C2BIT depends on the number of array accesses, not on the array size. (Otherwise the CNF size should be proportional to $N^2$.) The right plot shows that the solving time by MINISAT solver is basically consistent with CNF size for both CBMC and C2BIT.

Table 1 shows experiment results on 10 selected property verification problems on C programs. P3 to P9 are different properties on the same unrolled program. The rest are different properties on different programs. Col (Column) 2 and 3 show the number of lines of code after loop unrolling and the satisfiability of properties checked. The memory usage and solving time by MINISAT for CBMC and C2BIT+S are given in Col 4-5 and Col 6-7 respectively. Col 8 to 12 give results from C2BIT+S+R including total runtime (encoding+solving), number of refinement iterations, ratio of $\text{size}(\text{Mabs})/\text{size}(\text{MB})$ and runtime speedup over the setup without abstraction. C2BIT+S shows great improvements on memory usage and runtime compared to CBMC 2.4 on both SAT and UNSAT properties. One main reason is due to our efficient Boolean modeling of array variables and aggressive slicing. C2BIT+S+R shows speedup compared to C2BIT+S on all SAT properties because their satisfying solutions can be obtained in the under-approximate model with small encoding width.
For P2-P4 whose UNSAT cores are all small, C2BIT+S+R shows faster than C2BIT+S. For P1 whose UNSAT core is very big, Mabs is similar to MB in both size and solving time, while C2BIT+S+R needs extra time of constructing the abstraction. For P5, although the size of MB is 13 times bigger than that of Mabs, their solving time is similar because MINISAT itself is efficient enough in searching relevant constraints among redundant ones in this case. For P6, C2BIT+S+R is slower than C2BIT+S although the abstract model size is rather small. Since some constraints not in the UNSAT core may still be useful to assist the SAT solver to find conflicts quickly, the abstraction may increase the non-determinism by removing these constraints. Therefore there is a tradeoff between removing constraints to reduce model size and reducing solving time when the UNSAT core is not very small. Note for P9, the values in the braces indicate the results without the proposed improvement in computing updated encoding width that is even worse than without refinement. In summary, the slicing can greatly reduce memory usage and runtime, and the refinement is efficient to further reduce the solving time in cases that the satisfying solution for SAT properties can be found in the small encoding range of the under-approximation or that the UNSAT core is small for UNSAT properties.

5. Conclusions

We have presented a new approach that applies program slicing combined with compiler optimization to compute an accurate slice. We also explored the proof-based abstraction on our software model to further enhance the scalability. Experiments show that our technique can achieve significant speedups compared to the conventional BMC tool.

Acknowledgement: This research was supported in part by NSF grants 0519959, 0524052, NIJ grant 2005-IJ-CX-K017.

Reference