Towards Precise and Scalable Verification of Embedded Software

Malay K. Ganai, Aarti Gupta, Franjo Ivančić, Vineet Kahlon, Weihong Li,
Nadia Papakonstantinou, Sriram Sankaranarayanan and Chao Wang
NEC Laboratories America, Princeton, NJ, 08540
malay@nec-labs.com

***Abstract*— Inspired by the success of model checking in hardware verification and protocol analysis, there has been growing interest in research and development of tools for the automated verification of software. This paper provides an overview of our efforts towards achieving precise and scalable verification of embedded software in a model checking-based verification platform called F-SOFT. We combine the complementary strengths of static program analysis based on techniques such as abstract interpretation and model checking to ensure high efficiency with low false alarm rate. The model checker is also able to provide valuable debugging output in the form of witness traces, if a bug is discovered in the model of the program. We elaborate on techniques that enhance scalability of our bounded model checking (BMC) by communicating high-level model information from the source code to the back-end model checking engine. This includes model transformations that guide the model checker better, as well as a state-of-the-art SMT-based (Satisfiability Modulo Theories) BMC framework. Finally, we present several case studies that highlight the significance of the various steps in the overall F-SOFT flow.**

I. INTRODUCTION

Formal methods such as model checking and static program analysis are now routinely used in hardware and safety-critical embedded software, respectively. High costs of a faulty hardware/software incurred in repairs and damages, justify the additional up-front investment. However, with the widespread use of ever more complex embedded software, the development and maintenance costs are increasing substantially.

Model checking is an automatic technique for the verification of concurrent systems, and has several advantages over simulation, testing, and deductive reasoning. Inspired by the success of model checking in hardware verification and protocol analysis [1], there has been growing interest in research and development of model checking tools for the automated verification of software [2].

In practice, complementary verification approaches with different strengths are used to address the inherent scalability problem. Program analysis uses *incomplete* (but sound) methods such as numerical domain analysis, to provide correctness proofs for a pre-defined set of software errors. Such methods though scale well usually, they suffer from large false alarms. Model checking relies on use of finite models of the program (either through abstraction or by the restriction to finite control and data) so that it can explore the state space in an exhaustive and systematic manner to guarantee the validity of a generic temporal property specification. Furthermore, the model checker is able to provide valuable debugging output in the form of witness traces, if a bug is discovered in the model of the program. Model checking based methods, typically, do not scale with the model size, but are sound and complete with respect to the model. Understandably, both methods can be useful if combined suitably, but are too limited when applied individually to cope with the challenges of embedded software validation and verification.

We present a precise and scalable verification framework called F-SOFT [3] targeted for embedded system software written in C. We combine the above two approaches with recent static analysis and model checking advancements to address the scalability and low false alarm rate. It considers reachability properties for verification, in particular whether certain labeled statements are reachable in an automatic instrumentation of the program. It also includes checkers for a set of standard programming bugs such as array bound violations, string errors, null pointer dereferences, use of uninitialized variables, memory leaks, lock/unlock violations, division by zero, etc. These checkers are implemented by automatically adding property monitors to the given source code programs. Verification is performed via a translation of the given C program to a finite state circuit model, derived automatically by considering the control and data flow of the program (under the assumptions of bounded data and bounded recursion). The back-end model checking is performed by a tool called VERISOL [4], [5] (formerly, DIVER), which includes several state-of-the-art symbolic model checking techniques, geared towards software verification [6], [7].

II. BOUNDED MODEL CHECKING

Bounded Model Checking (BMC) [8] has been successively applied to verify real-world designs, and provides a more scalable verification solution compared to BDD-based symbolic model checking [9]. BMC is a model checking technique where the falsification of a given LTL property is checked at a given sequential depth. Typically, it consists of the following steps: unrolling of the design for the given number of time frames, translating the BMC instance into a decision problem ϕ such that ϕ is satisfiable *iff* the property has a counter-example of depth (less than or) equal to k, and using a decision procedure to check if the problem is satisfiable. In SAT-based BMC, ϕ is a propositional formula, and the decision procedure is a Boolean SAT solver. Though several state-of-art advancements have been proposed in SAT-based BMC [5], there are inherent disadvantages in using a Boolean representation. Propositional translations of richer data types, e.g., integer, and high-level expressions, linear arithmetic, lead to large bit-blasted formulas. Note, it is often useful to perform range analysis to determine adequate bounds for the integer data values. Moreover, the high-level semantics are often "lost" in such low-level translation, thereby, the SAT search becomes more difficult. 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 Satisfiability Modulo Theory (SMT) solvers [10], [11], [12], [13] and SMT-based verification methods, using richer expressive theories beyond Boolean logic. Model checking methods such as SMT-based BMC [14], [15], [16], [6] overcome many of the limitations of SAT solvers applied on Boolean expressions. In SMT-based BMC, the BMC problem ϕ is translated into a quantifier-free formula (QFP) in a decidable subset of first order logic, which is then solved by SMT solver. With recent advancements in SMT solvers built over DPLL style SAT solvers [11], [10], SMT-based verification methods look quite

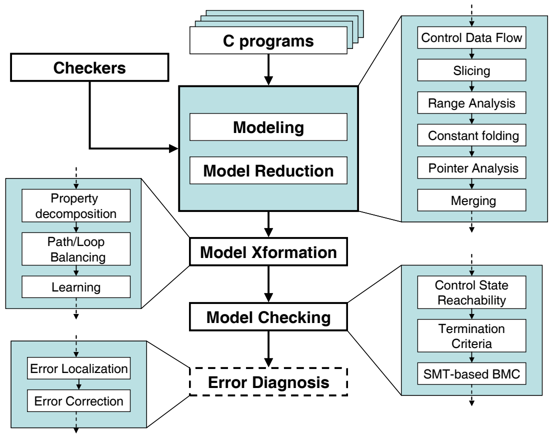


Fig. 1. F-SOFT Tool Overview

promising in providing more scalable alternatives than SAT-based or BDD-based methods. Further, with property preserving model transformation, and the use of static control flow information [6], SMT-based BMC can find deeper bugs, in comparison to SAT-based BMC.

III. VERIFICATION OF SOFTWARE PROGRAMS: OVERVIEW

We describe our software verification tool F-SOFT [3], shown in Figure 1. Our verification procedure can be viewed as comprising three phases: (I) Generate a reduced model of the software that includes monitors, if required, to check properties of interest, (II) Transform the model to suit the verification back-end, and (III) Verify properties using state-of-the-art SAT/SMT-based BMC.

As part of *Phase I*, we first use CIL [17] in the frontend to make all expressions side-effect-free (adding temporary variables as needed), to make all identifiers globally unique, and to rewrite complex C constructs in terms of simpler ones (e.g. `switch` and `for` in terms of `if` and `goto`). Then, software modeling is performed to extract a finite state model, which is represented using *Boolean* and *arithmetic* expressions (Section IV). We instrument the model with a set of monitors to check correctness properties. Next we reduce the model using abstract interpretation and various other program analysis techniques (Section V). Specifically, we use program slicing, constant folding, range analysis [18], and merging to reduce the model size. We also use several sound numerical analysis techniques [19], [20] to conservatively prove the correctness of many properties. A reduction in the set of properties implies further scope for model reduction using program slicing. Since these static techniques are quite efficient and thus have a relatively low overhead compared to model checking, we generally try to resolve as many properties as possible in this phase of the analysis. For a typical null pointer dereference checker, we can resolve 50-80% properties by static analysis alone. Though several such properties get resolved during this phase, many of them still remain unresolved. To reduce the burden on the designer, we apply BMC to find precise witness traces that can guide the debugging effort of the user.

To provide a robust and scalable BMC, we apply property preserving modeling transformation, in *Phase II*, to obtain "verification-friendly" models [6], [21]. We also decompose set of the properties and use property-based slicing to improve the effectiveness of analysis on resulting smaller models. We extract high-level information such as invariants, control flow graphs and program semantics, which are used for *guidance* in the following phase. These model transformations have been found to be very effective in improving the performance of BMC (Section VI).

In *Phase III*, we use SMT-based BMC framework to exploit the high-level information obtained from the previous phase to reduce the BMC instance sizes. We use SMT solvers [22], [23], [24], [25] instead of a traditional SAT solver, to exploit the richer expressiveness, in contrast to bit-blasting. Furthermore, in a typical verification scenario with multiple properties to resolve, it is often not clear how to devise a good verification procedure, i.e., how to balance the limited time resources between proving or falsifying the correctness properties. We have devised cheaper-to-compute termination criteria for BMC when applied to commonly-used programs [7] (Section VII). If a true counterexample is discovered, a testbench program (in C) is automatically generated, which can be executed in the user's favorite debugger for analyzing the trace. As part of ongoing work, we are adding support for error diagnosis, comprising techniques for error localization and error correction.

Using several industry case studies, we describe our verification procedure highlighting the significance of each step in the flow. We also compare with contemporary practices and evaluate our approach against them (Section VIII).

A product based on F-SOFT is now being used internally in our company to find bugs in production-level software. Due to its flexible and modular infra-structure, F-SOFT provides a very productive verification platform for research and development.

IV. BUILDING MODELS FROM C

We briefly discuss our model building step (details in [3]) from a given C program under the assumption of a bounded heap and a bounded stack. We begin with full-fledged C and apply a series of source-to-source transformations into smaller subsets of C, until program state is represented in a simplified control flow graph (CFG) as a collection of simple scalar variables of simple types (Boolean, integer, float) and each program step is represented as a set of parallel assignments to these variables. We handle pointer accesses using direct memory access on a finite heap model, and apply standard slicing constant propagation, and points-to analysis [26].

We do not inline non-recursive procedures to avoid blow up, but bound and inline recursive procedures. Each non-recursive function is inlined exactly once; function calls are replaced with `goto`s to the function's first statement. Parameters and return values are passed via global variables, by adding assignments at each function call. Function return is handled by storing a unique id of the call site in a global variable before the call, and replacing `returns` with groups of `gotos` conditioned on this variable. An alternative is to inline each function at each call site [27], but this can significantly increase the model size. The C program now consists of labeled blocks of assignments followed by conditional `gotos`, corresponding a CFG. From the simplified CFG, we build an EFSM[1] where each block is identified with a unique *id* value, and a control state variable PC denoting 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 add an update transition

[1]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 linear space $D_1 \times \ldots \times D_n$ (each point in D is described by the 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 (program) *state* of M. A SINK (SOURCE) state is a unique control state with no outgoing (incoming) transition.

relation based on the expressions assigned to the variable in various basic blocks in the CFG. We use *Boolean* and *arithmetic* expressions to represent the update and guarded transition functions. We rewrite the assignments in each block so that they can be executed in parallel while preserving the sequential semantics; for instance, in the sequence {`x=y; z=x+2`} the second assignment is rewritten as {`z=y+2`}. Then, we rewrite the entire program as a group of iterated parallel assignments. The common design errors such as array bound violations, null-pointer deferences, and assertion violations are modeled as ERROR blocks. We focus on the reachability of such ERROR blocks.

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

Fig. 2. A sample C code with function calls

In Figure 2, we present a sample C program, and its corresponding EFSM M obtained by our modeling is shown in Figure3. Each box represents a control state (or basic block) and the unique number in the attached square denotes its block *id*. For example, the edge $(4, 5)$ represents a transition from block 4 to 5 is 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 $[2, 9]$, $[3, 8]$, and $[14, 10]$ correspond to *call* and *return* sites for `bar`. The variable `cxt_id` identifies the various contexts i.e., call/return sites.

V. Model Reduction: Static Analysis

Numerical domain static analysis has been used to prove the runtime safety of programs for properties such as the absence of buffer overflows, null pointer dereferences, division by zero, string library usage and floating point errors [28], [29], [30]. Domains such as intervals, octagons and polyhedra are used to symbolically over-approximate the set of possible values of integer and real-valued program variables along with their inter-relationships under the abstract interpretation framework [31], [32], [33], [34], [35], [36], [37]. These domains are classified by their precision, i.e, their ability to represent sets of states and tractability, the complexity of common operations such as union (join), post-condition, widening and so on. In general, enhanced precision leads to more proofs and less false positives, while resulting in a costlier analysis.

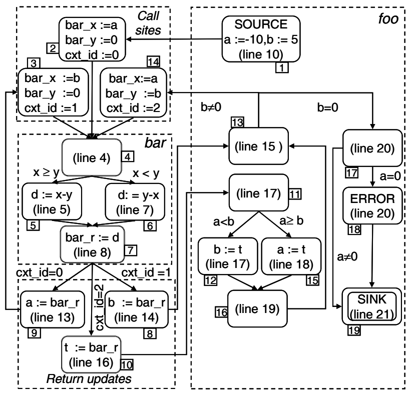


Fig. 3. EFSM of the sample code

On the simplified CFG instrumented with ERROR blocks, we perform various static analyses to simplify the model and focus our search on the potential alarms raised by the static analyzer. By employing a sequence of static analyses, one building upon the other, we hope to find many simple proofs for many properties. Common static analyses that we employ include program slicing, constant folding, range analysis [35], [32], [18], and abstract interpretation [38] on numerical domains such as the the octagon domain [34], [19] and polyhedral domain [39], [20]. Some details are described below.

We efficiently determine conservative value ranges of program variables by performing range analysis. Our main method [18] is based on the framework suggested in [40] which formulates each range analysis problem as a system of inequality constraints between symbolic bound polynomials. It then reduces the constraint system to an LP (linear programming) problem, which can be analyzed by any available LP solver. The solution to the LP problem provides interval values of all variables. In other words, for each variable x, we obtain the upper and lower bound such that $l_x \leq x \leq u_x$, where l_x, u_x are integer constants.

Octagonal invariants are invariants of the form $\pm x \pm y \leq c$, where x and y are integer program variables and c is an integer constant. These invariants can be computed efficiently by the octagon abstract domain [34], [19]. The octagon abstract domain has been used within Astrée [29], and was shown instrumental in reducing the number of false alarms when detecting runtime errors in critical embedded software [41].

For properties that may need constraints of a more complex form than octagon domain, we employ a more precise analysis (and costlier) based on the polyhedral domain [20]. Specifically, for a variable x, symbolic range constraints are used, i.e., $f_l^x \leq x \leq f_u^x$, where f_l^x, f_u^x are polynomial functions of other program variables. Such symbolic bounds have been found useful in proving more properties in real code than octagon domain does not, with some more overhead.

To show the relative advantages of these static proof techniques, we experimented on two industry example sets, with total of 480 and 15878 error reachability properties corresponding to array bound violations, respectively. The interval analysis (with constant folding) proves many of the properties including the common case of static arrays accessed in loops with known bounds. While the octagon domains and symbolic ranges can complete on all the examples even in the absence of such simplifications, we run interval analysis as a first step. Using interval range analysis alone, we are able to resolve more than 60% properties. Additional 5-10% proofs are obtained using octagon and symbolic range analysis on the model reduced after removing the proved properties. The overhead of precise analyses are found to be small and quite affordable on these set of examples. For set 1, complete analysis takes around 2 minutes (= 10s (interval analysis) + 29s (octagona) + 81s (symbolic)), with 30s per proof on average. For set 2, complete analysis takes around 4 hours (= 3.6 hr (interval analysis) + 180s (octagon) + 439s (symbolic)), with 1s per proof on average.

In addition to proving properties that do not require further scrutiny in our framework, static analysis allows us to remove error blocks corresponding to such properties. We apply property-based slicing to further simplify the model post-static analysis. Further, we use the value and symbolic ranges for restricting the possible values in state space search during model checking. Specifically, we use octagon

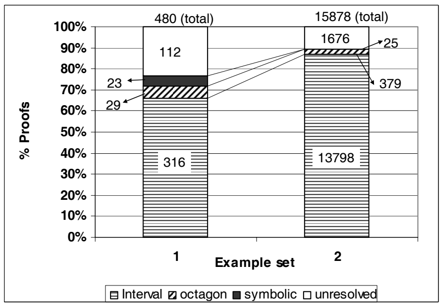


Fig. 4. Static Proofs

invariants to strengthen [19], [6] the concrete transition relation at a given program location (block) l by conjoining it with the octagonal invariants that hold at the program location l (Section VII).

VI. Verification "Aware" Model Transformation

We have proposed the use of model transformation in the context of synthesis-for-verification (SFV) methodology to improve the verification performance [21]. As part of the SFV methodology, we identified various modeling techniques which results in "verification-friendly" models that are relatively easier to model check. Such SFV paradigm of generating verification aware models has been found very effective in practice [6]. To improve the performance of SMT-based BMC, we discuss a control-flow based model transformation that preserves the validity of the model with respect to the property checked.

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 Figure 2), we obtain the set $R(d)$ for $0 \leq d \leq 7$ as follows: $R(0) = \{1\}$, $R(1) = \{2\}$, $R(2) = \{5, 6\}$, $R(3) = \{7\}$, $R(4) = \{8, 9, 10\}$, $R(5) = \{13, 3, 11\}$, $R(6) = \{14, 17, 4, 12, 15\}$, $R(7) = \{4, 18, 19, 5, 6, 16\}$.

CSR can be used to reduce the size of BMC instances [6]. Basically, if a control state $r \notin R(d)$, then the unrolled transition relation of variables that depend on r can be *simplified*, as described in the following. We define a Boolean predicate $B_r \equiv (PC = r)$, where PC is the program counter that tracks the current control state. Let v^d denote the unrolled variable v at depth d during BMC unrolling. Consider again model M in Figure 2, where the next state of variable bar_x is defined as $next(bar\_x) = (B_2 || B_{14})\ ?\ a\ :\ B_3\ ?\ b\ :\ bar\_x$; (using the C language notation for *cascaded if-then-else*). For depths $2 \leq k \leq 4$, blocks $2, 3, 14 \notin R(k)$, and therefore, $B_2^k = B_3^k = B_{14}^k = 0$. Using this unreachability control state information, we can *hash* the expression representation for bar_x^{k+1} to the existing expression bar_x^k, i.e., $bar\_x^{k+1} = bar\_x^k$. This hashing, i.e., reusing of expression, considerably reduces the size of the logic formula, i.e., the BMC instance. Note, 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 saturation of CSR [6]. Typically, saturation of CSR leads to large $|R(d)|$, and adversely affects the size of the unrolled BMC instances.

To avoid saturation, we proposed a model transformation strategy called *Balancing Re-convergence* or *Path/Loop Balancing* (PB) [6]. It transforms an EFSM by inserting NOP states such that lengths of the 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. As an example, consider the CFG shown in Figure 5(a) (taken from [6]). It has 3 loops with backedges $(6, 3)$, $(8, 3)$ and $(7, 1)$, respectively. CSR on this CFG saturates at depth 6 with $|R(6)| = 8$. By using the PB strategy, the re-converging paths between 1 and 3, 3 and 5, and 4 and 6 are balanced by inserting nodes shown as unshaded circles in Figure 5(b). Similarly, the loops are balanced by inserting new nodes in the backedges so that their periods are matched. Note, the number of nodes to be inserted is determined by computing the weights of the re-convergent paths as described in [6]. CSR on the CFG shown in Figure 5(b) does not saturate, and max $|R(d)| = 4$. Such PB techniques have shown to be very effective in accelerating BMC [6].

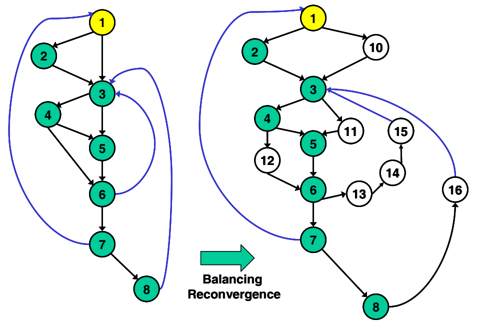
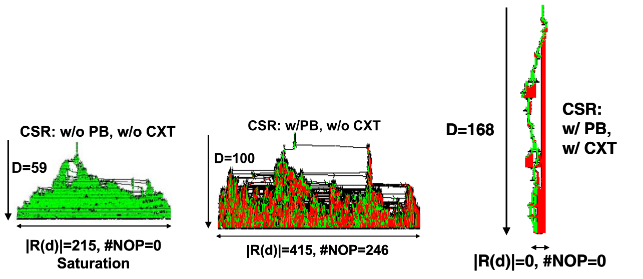


Fig. 5. CFG transformation using PB

PB techniques are applicable [6] only when the CFG is *reducible* [42]. A reducible graph 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 Figure 2, is not reducible, although the program in Figure 2 is *well-structured*, i.e., it has only a reducible loop. The CSR on the CFG leads to saturation and therefore, is not effective in accelerating BMC. One cause for irreducibility of CFG is the introduction of unstructured loops during modeling, which are not present in the original program. For example (in Figure 2), the loop $3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 3$ is an unstructured loop, and is not present in the original program (Figure 2). Such false loops are introduced due to non-inlining of functions. A function *return*, denoted by edge $(7, 9)$, does not correspond to the function *call*, denoted by the edge $(3, 4)$. We overcome this problem by making PB strategy context-sensitive. (One can choose to inline function calls, but it may result in a blow-up in size of EFSM.) Similarly, we make the CSR context-sensitive; otherwise, many false paths through CFG will make $R(d)$ large.

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 performed for the following cases: (a) `CSR`: model with no `PB` and 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 Figure 6(a)-(c) up to depth D. The width of



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

Fig. 6. Combining CSR with PB/CXT

the graph, proportional to $|R(d)|$ where $0 \leq d \leq D$, indicates the scope of BMC simplification. We observe that using method CSR, $R(d)$ saturates at depth $D = 59$; CSR+PB does not saturate but $R(d)$ is significantly larger compared to the method CSR+PB+CXT. Thus, the method *CSR+PB+CXT* has the largest potential to improve the performance of BMC.

VII. Verification Engine

We present the flow of our approach for SMT-based BMC as shown in Figure 7. Given an EFSM Model M and a property P (box 0), we perform a series of property preserving transformations (box 1). After that we perform context-sensitive CSR on the transformed model (box 2). Using the reachability information, we generate simplification constraints on-the-fly at each unroll depth k (Section VII-B, box 3). These simplification constraints are used by the expression simplifier (Section VII-A box 4) during unrolling to reduce the formula. These constraints are also used to improve the search on the translated problem. We also use incremental learning technique (box 15) i.e., re-use of theory lemmas in SMT-based BMC framework. We present various anlaysis modules in the order of ease of explanation.

A. Expression Simplifier

High-level expressions in our framework include Boolean expressions *bool-expr* and term expressions *term-expr*. Boolean expressions are used to express Boolean values *true* or *false*, Boolean variables (*bool-var*), propositional connectives $(\vee, \wedge, \neg)$ relational operators $(<, >, \leq, \geq, ==)$ between term expressions, and *uninterpreted predicates* (UP). Term expressions are used to express integer values (*integer-const*) and real values (*real-const*), integer variables (*integer-var*) and real variables (*real-vars*), linear arithmetic with addition (+) and multiplication (*) with *integet-const* and *real-const*, *uninterpreted funtions* (UF), *if-then-else* (ITE), *read* and *write* to model memories. To model behavior of a sequential system, we also have a *next* operator to express the next state behavior of the state variables.

Our high-level design description is represented in a semi-canonical form using an expression simplifier. The simplifier rewrites expressions using local and recursive transformations in order to remove structural and multi-level functionally redundant expressions, similar to simplifications proposed for Boolean logic [43] and also for first order logic [44]. Our expression simplifier has a *compose* operator [45], that can be applied to unroll a high-level transition relation and obtain on-the-fly expression simplification; thereby achieving simplification not only within each time frame, but also across time frames during unrolling of the transition relation in BMC.

B. Simplification Constraints

At any unrolling depth d of BMC, we apply the following on-the-fly structural and clausal (learning-based) simplification on the corresponding formula [6]. Note, these simplifications are effective for small $|R(d)|$.

- *Unreachable Block Constraint (UBC):* If a control state r is not reachable at depth d, the predicate $B_r \equiv (PC = r)$ will evaluate to FALSE at depth d. We simplify the formula by propagating $B_r = 0$ at depth d.
- *Reachable Block Constraint (RBC):* At any depth d, at least one control state in $R(d)$ is reachable.
- *Mutual Exclusion Constraint (MEC):* At any depth d, at most one control state in $R(d)$ is the current state.
- *Forward Reachable Block Constraint (FRBC):* At any depth d, if current control state is r i.e. $B_r^d = TRUE$, then the next state must be among the $from(r)$ set, where $from(r)$ is the set of control states reachable from r in one step.
- *Backward Reachable Block Constraint (BRBC):* At any depth $d > 0$, if current state is r, i.e., $B_r^d = TRUE$, then the previous state at depth $d-1$, must be among the $to(r)$ set, where $to(r)$ is the set of control states reachable to r in one step.
- *Block-Specific Invariant (BSI):* At any depth d, a given invariant C_r for a given state r is valid only if r is the current state at depth d. We obtain C_r from static numerical domain analysis (Section V).

C. BMC Termination Criteria

BMC, in general, is incomplete unless checking is performed up to the completeness threshold ($\mathcal{CT}$) bound [8], [46], [47]. In general, computing $\mathcal{CT}$ bound is computationally expensive. For a safety property $\mathbf{G}p$ (where p is a non-temporal expression), optimum $\mathcal{CT}$ is shown [8] 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 (LFP), by computing a series of SAT checks with increasing k [8]. Such computation requires solving SAT instances of size $O(k^2)$. Thus, each LFP check for computing rrd grows quadratic in size and becomes harder to solve with unrolling. 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-optimum.

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. 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 [48]. We devised efficient proof techniques geared for terminating software programs in an SMT-based BMC framework. We proposed 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 showed that for a terminating program, the length of the longest NTP corresponds to the recurrence diameter of the corresponding EFSM [7].

D. SMT-based BMC

We describe the flow of SMT-based BMC with NTP checks, shown in Figure 7. Note that the flow is applicable to both terminating and non-terminating programs; however, we will not obtain a $\mathcal{CT}$ bound for the latter. Note, we avoid an expensive LFP check that

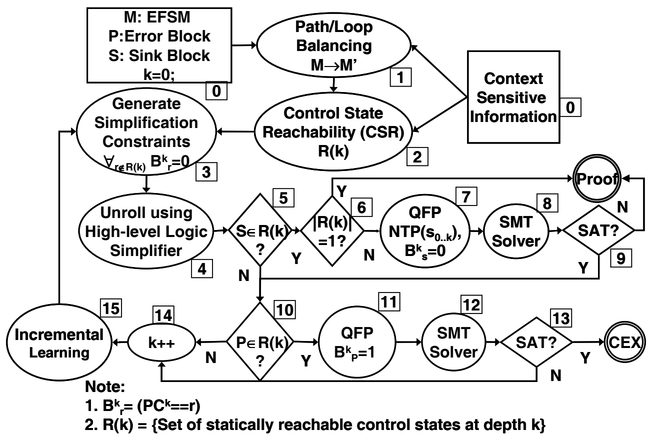

M: EFSM
P:Error Block
S: Sink Block
k=0;
0
Path/Loop Balancing M→M'
1
Context Sensitive Information
0
Generate Simplification Constraints $\forall_{r \notin R(k)} B^k_r=0$
3
Control State Reachability (CSR) R(k)
2
Proof
Unroll using High-level Logic Simplifier
4
5
S∈R(k) ?
6
|R(k)| =1?
7
QFP NTP($s_{0..k}$), $B^k_s=0$
8
SMT Solver
SAT?
9
Y
N
N
Y
N
Y
15
Incremental Learning
14
k++
10
P∈R(k) ?
11
QFP $B^k_P=1$
12
SMT Solver
13
SAT?
CEX
N
Y
Y
N
Note:
1. B^k_r= (PC^k==r)
2. R(k) = {Set of statically reachable control states at depth k}

Fig. 7. SMT-based BMC

hardly ever succeeds in practice for non-terminating programs. We describe the SMT-based BMC flow, where each step number matches the tagged block in Figure 7.

- **Step 1:** Given *EFSM* M, with an ERROR block P and a SINK block S, carry out context-sensitive PB to transform the model to M'.
- **Step 2:** Generate context-sensitive CSR information $R(k)$ of the CFG of M'.
- **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. 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.

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). Using PB and context-sensitive CSR (Section VI), we also reduce static reachability $SINK$ states, and hence reduce the number of NTP checks. As shown in Figure 6, we observe that using method CSR, $SINK$ block appears *every step* after saturation; using method CSR+PB, it appears *every other step*; and using method CSR+PB+CXT, it appears only *once*. Thus, the method CSR+PC+CXT also has a better proof finding capability.

VIII. Experiments

In our first set of experiments, we demonstrate the role of model transformation and learning in model checking on an industry example. In the second set of experiments, we provide results of our verification analysis engines, and compare it with a related tool CBMC [27].

TABLE I
Effect of Model Transformation and Learning

P	I: M+No Learning			II: M+Learning			III: M'+Learning		
	D	sec	W?	D	sec	W?	D	sec	W?
P1	9*	TO	N	38*	TO	N	41	1	Y
P2	9*	TO	N	41*	TO	N	44	1	Y
P3	9*	TO	N	43*	TO	N	92	156	Y
P4	9*	TO	N	30	188	Y	94	151	Y
P5	9*	TO	N	21	6	Y	60	4	Y
P6	9*	TO	N	31	164	Y	70	22	Y

A. Model Transformation Results

To show the effect of model transformation (Section VI) and learning (Section VIIB), we experimented on industry software written in C with about 17K lines of code. We first generated an EFSM model M with 259 control states and 149 state (term) variables. The data path elements include 45 adders, 987 if-then-else, 394 constant multipliers, 53 inequalities, 501 equalities and 36 uninterpreted functions. The corresponding flow graph has 12 natural loops. We consider reachability properties P1-P6 corresponding to six control states. CSR on M saturates at depth 84. After transforming M using path and loop balancing algorithms, we obtain a model M' with 439 control states and max loop period N=4. We ran SMT-based BMC for 500s (for each of P1-P6) on: (I) Model M without learning (using only expression simplification), (II) Model M with learning, and (III) transformed Model M' with learning. We present our results in Table I. Column 1 shows the property checked; Columns 2-4 report BMC depth reached (* denotes depth at time out, TO), time taken (in sec) and whether witness was found (Y/N), respectively, for combination (I). Similarly, Columns 5-7 and 8-10 present information for combinations (II) and (III), respectively. The results clearly show that combination (III) is superior to (II) and (I), with significant improvement in the performance, though at increased witness depth.

B. Verification Results

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, `t0-t8` is an air traffic control and avionic system with *assertions* checks; `f` is a restart module of *wu-ftpd* with *array bound violations* checks; `m1-m2` examples are for a network protocol with *null pointer de-references* checks; and `h1-h4` examples correspond to software for cell phones with *array bound violation* checks.

Our experiments were conducted on a workstation with 3.4GHz, 2GB of RAM running Linux, with a time limit of 1000s for each run. In practice, verification engineers have to run several examples, and they typically allocate 10-20 minutes for each example. We first apply static proof techniques (SA), and resolve as many properties as possible statically, and thereby, reduce the model size. In the next phase, we apply model transformation and learning, followed by SMT-based BMC on the unresolved properties. 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 perform a controlled experiment with various strategies, and show the BMC comparison results in Table II.

Column 1 shows the name of the benchmark with number of lines of code (LoC), Column 2 shows the total number of checkers

TABLE II

VERIFICATION RESULTS: COMBINING STATIC ANALYSIS WITH SAT/SMT BMC

Ex (LoC)	**#prp**	**SA Proof**		**BMC with(+)/without(–) SMT, SAT, PB: Path/Loop Balance, CXT: ConteXT-sensitivem LFP: Loop-Free Path Check, NTP: Non-Terminating Path Check (P≡# Proofs, W≡# Witnesses, TO≡Time-out, D≡BMC Depth, MO≡Mem-out)**													
				`-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	sec	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
t0 (445)	1	0	0.3s	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
t1 (445)	1	0	0.3s	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
t2 (445)	1	0	0.3s	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
t3 (445)	1	0	0.3s	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
t4 (445)	1	0	0.3s	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
t5 (445)	1	0	0.3s	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
t6 (445)	1	0	0.3	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
t7 (445)	1	0	0.3s	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
t8 (445)	1	0	0.3s	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
ng4 (135)	14	11	2s	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 (114)	9	7	2s	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
h1 (2K)	144	132	24s	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
h2 (2K)	158	144	26s	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
h3 (389)	116	59	57s	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
h4 (2K)	188	155	274s	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
m1 (423)	25	3	0.2s	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
m2 (2.5K)	104	16	1s	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
f (839)	26	21	2s	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

added, and Columns 3-4 show the number of static proofs and time required, respectively. Columns 5-18 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` denote CSR without context-sensitive analysis. To illustrate, Columns 5-6 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 5 shows number of proofs (P), witnesses found (W) and unresolved properties (`P/W/?`), and Column 6 shows number of BMC unrollings (D) performed with time (in sec) in parenthesis. As an example, for `m1` with 423 LoC and 25 total checkers, `SA` proved 3 properties taking less than a sec. For the remaining 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 report number of NTP checks ($\#NTP$) in Columns 15 and 18, 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+SAT` as our baseline [3]. 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.

C. Comparison with Related Tools

We also performed experiments on these examples using CBMC [27], a SAT-based BMC for verifying C programs. For fair comparison, we used the post-static analysis models and property sets (i.e., after applying `SA`). CBMC could not resolve any property in any of these examples, due to mem-out. In contrast, the model checking engine in F-SOFT is able to resolve 157 out of 242 (=65%) properties within the given resource limitations. Such a result is not surprising, given many differences between CBMC and F-SOFT. In CBMC [27], given a BMC bound, the C program is transformed into an equivalent static single assignment (SSA) form with bounded loop and recursion, which is then bit-blasted to derive a Boolean SAT formula. In F-SOFT a finite state model (not just a formula) is generated from the C program, without unwinding loops and without multiple function inlinings. Our model building approach, as opposed to directly generating formula, also help us to apply light-weight static analysis to reduce the model and the property set. Further, using the model transformation and simplification constraints during unrolling we are able to reduce the BMC instance sizes at each depth. These techniques allow F-SOFT to scale better than CBMC on larger programs, as also demonstrated by our experimental results.

IX. CONCLUSION

We presented an overview of our efforts in combining precision of model checking with scalability of light weight static analysis. Each analysis targets capacity and performance issues inherent in verifying complex software. Use of static program analysis and model transformation are key to the success of model checking tools. Using several industry examples, we described the interplay of these engines highlighting their contributions at each step of verification.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the support and assistance of Yuusuke Hashimoto-san, Kenjiroh Ikeda-san, Shinichi Iwasaki-san, and the NEC SWED team during the development of F-SOFT.

REFERENCES

[1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[2] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. of CAV*, 2001.

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

[4] M. K. Ganai, A. Gupta, and P. Ashar. DiVer: SAT-based model checking platform for verifying large scale systems. In *Proc. of TACAS*, 2005.

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

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

[7] M. K. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In *Proc. of DATE*, 2008.

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

[9] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
[10] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV*, 2006.
[11] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propogation and its application to difference logic. In *Proc. of CAV*, 2005.
[12] C. Barrett, D. Dill, and Jeremy Levitt. Validity Checking for Combinations of Theories with Equality. In *Proc. of FMCAD*, November 1996.
[13] 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.
[14] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. of CADE*, 2002.
[15] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. of CAV*, 2002.
[16] 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.
[17] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.
[18] A. Zaks, I. Shlyakhter, F. Ivančić, S. Cadambi, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Using range analysis for software verification. In *International Workshop on Software Verification and Validation*, 2006.
[19] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *Proc. of CAV*, 2006.
[20] S. Sankaranarayanan, F. Ivančić, and A. Gupta. Program analysis using symbolic ranges. In *Proc. of SAS*, 2007.
[21] M. K. Ganai, A. Mukaiyama, A. Gupta, and K. Wakabayashi. Synthesizing "verification aware" models: Why and how? In *Proc. Intl. Conf. on VLSI*, 2007.
[22] C. Wang, F. Ivančić, M. K. Ganai, and A. Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. In *Proc. of Logic Programming and Automated Reasoning*, 2005.
[23] M. K. Ganai, M. Talupur, and A. Gupta. SDSAT: Tight Integration of Small Domain Encoding and Lazy Approaches in a Separation Logic Solver. In *Proc. of TACAS*, 2006.
[24] C. Wang, A. Gupta, and M. K. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *Proc. of DAC*, 2006.
[25] SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.
[26] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of PASTE*, 2001.
[27] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS*, 2004.
[28] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of Network and Distributed Systems Security Conference*, 2000.
[29] B. Blanchet, P. Cousot, R. Cousot, J. Ferret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of PLDI*, 2003.
[30] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proc. of PLDI*, 2003.
[31] M. Karr. Affine relationship among variables of a program. In *Acta Inf.*, 1976.
[32] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming*, 1976.
[33] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *Proc. of POPL*, 1978.
[34] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, 2001.
[35] S. Sankaranarayanan, M. Colon, H. Sipma, and Z. Manna. Efficient strongly relational polyhedra analysis. In *Proc. of VMCAI*, 2006.
[36] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proc. of SAS*, 2004.
[37] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proc. of SAS*, 2006.
[38] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximate of fixpoints. In *Proc. of POPL*, 1977.
[39] Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. In *Theor. Comput. Sci.*, 2005.
[40] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proc. of PLDI*, 2000.
[41] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, 2004.
[42] A. B. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-wesley Publishing Company, 1988.
[43] M. K. Ganai and A. Kuehlmann. On-the-fly compression of logical circuits. In *Proc. Intl. Workshop on Logic Synthesis*, 2000.
[44] J.-C. Filliatre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated canonizer and solver. In *Proc. of CAV*, 2001.
[45] M. K. Ganai and A. Aziz. Improved SAT-based bounded reachability analysis. In *Proc. of VLSI Design Conference*, 2002.
[46] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT solver. In *Proc. of FMCAD*, 2000.
[47] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Proc. of FMCAD*, 2000.
[48] P. Cousot and R. Cousot. Verification of embedded software: Problems and perspectives. *LNCS*, 2001.