Accelerating High-level Bounded Model Checking

Malay K Ganai and Aarti Gupta
{malay | agupta } at nec-labs dot com
NEC Laboratories America, Princeton, NJ USA 08540

Abstract: *SAT-based Bounded Model Checking (BMC) has been found promising in finding deep bugs in industry designs and scaling well with design sizes. However, it has limitations due to requirement of finite data paths, inefficient translations and loss of high-level design information during the BMC problem formulation. These shortcomings inherent in Boolean-level BMC can be avoided by using high-level BMC. We propose a novel framework for high-level BMC, which includes several techniques that extract high-level design information from EFSM models to make the verification model "BMC friendly", and use it on-the-fly to simplify the BMC problem instances. Such techniques overcome the inherent limitations of Boolean-level BMC, while allowing integration of state-of-the-art techniques for BMC. In our controlled experiments we found signficant performance improvements achievable by the proposed techniques.*

1. Introduction

To cope with the increasing design complexity and demand to reduce design cycle time, the focus has shifted towards supporting high-level design abstraction, synthesis and verification methodologies. At the Boolean-level of design representation, SAT-based Bounded Model Checking (BMC) [1-4] due to several advancements — improved DPLL-style SAT solvers [5], on-the-fly circuit simplification [6, 7], and SAT-based incremental learning [3, 7, 8] — has been gaining wide acceptance as a scalable verification solution compared to BDD-based symbolic model checking [9]. With advent of sophisticated SMT solvers [10-15] built over DPLL-style SAT solvers, SMT-based BMC [16, 17] is also gaining popularity. Unfortunately, we do not see a similar level of maturity and advancements in verification efforts at higher levels of abstraction. This is mainly due to higher theoretical complexity, and a wide engineering gap between theoretical and practical solutions at the higher levels. To reduce this gap, we propose a framework to efficiently perform high-level BMC using SMT (Satisfiability Modulo Theory) solvers that overcome the inherent limitations of SAT-based Boolean-level BMC, while allowing integration of state-of-the-art techniques adopted for Boolean-level BMC. In this framework, we apply three novel techniques to accelerate high-level BMC (as shown in Figure 1):

- efficient extraction of high-level information,
- its use to obtain a "BMC friendly" verification model through model transformations, and
- its on-the-fly application during BMC to simplify BMC problem instances.

ICCAD 2006, November 5-9, 2006, San Jose, USA.

1.1. Bounded Model Checking

BMC is a model checking technique where falsification of a given LTL property is checked for a given sequential depth, or *bound* [1, 2]. Typically, it involves three steps:

- The design with the property f is unrolled for k (bounded) number of time frames.
- The BMC problem is translated into a propositional formula φ such that φ is satisfiable *iff* the property f has counter-example of depth (less than or) equal to k.
- A SAT-solver is used for the satisfiability check.

1.2. Boolean-level BMC and its Limitations

In Boolean-level BMC, the translated formula is expressed in propositional logic and a Boolean SAT solver is used for checking satisfiability of the problem. Several state-of-the-art techniques [18] exist for Boolean BMC that have led to its emergence as a mature technology, widely adopted by the industry. However, there are several limitations of a propositional translation and use of a Boolean SAT Solver. Some of these are as follows:

- A propositional translation in the presence of large data-paths leads to a large formula; which is normally detrimental to a SAT-solver due to increased search space.
- Data-path sizes need to be known explicitly *a priori,* before unrolling of the transition relation. For unbounded data-path, additional range-analysis of the program/design is required to obtain conservative but finite data-path sizes.
- High-level information is lost during Boolean translation and therefore, needs to be re-discovered by the Boolean SAT solver often with a substantial performance penalty.

1.3. High-level BMC

High-level BMC overcomes the above limitations of a Boolean-level SAT-based BMC; wherein, a BMC problem is translated typically into a quantifier-free formula in a decidable subset of first order logic, instead of translating it into a propositional formula; the first order logic formula is then solved by a high-level solver, such as an SMT solver.

In [12], an expressive logic called CLU (counter arithmetic logic with lambda-expressions and uninterpreted functions) is used to model systems. The decision procedure is based on a hybrid procedure using either a small model instantiation with conservative ranges or a predicate-based encoding. It generates an equi-satisfiable Boolean formula, which is then checked using a Boolean SAT solver. In [16, 17], the expressive logic used is linear arithmetic (addition and multiplication by constants), arrays, records, lists, bit-vectors; where SMT solvers are used to check the satisfiability. Note that these previous approaches [12, 16, 17] overcome part of the limitations of a

Boolean-level SAT-based BMC as discussed above, but lose some or all of the features that state-of-the-art Boolean-level BMC approaches provide.

Outline In Section 2 we give an overview of our contributions, in Section 3 we give relevant background on EFSM and flow graphs; in Section 4 we discuss our approach in detail; in Section 5 and 6 we discuss our experiments; and in Section 7 we conclude with summarizing remarks.

2. Our Contributions

We propose several methods to efficiently perform high-level BMC using an SMT solver that not only overcome the inherent limitations of SAT-based BMC but also allow integration of state-of-the-art innovations [18] adopted for the latter. Our high-level problem description uses a decidable quantifier-free fragment of first-order logic, including Presburger arithmetic, uninterpreted functions/predicates, arrays. Specifically, in our high-level BMC framework:

1. We use *expression simplification* to reduce the size of the unrolled formula not only within a time-frame, but across time-frames also.
2. We efficiently *extract* high-level information such as control-flow of the program/design.
3. We use the high-level information to *simplify and reduce* the unrolled formula size.
4. We provide *on-the-fly relevant* high-level information at each unrolling to the high-level solver, thereby not unduly overburdening the solver.
5. We use incremental learning, i.e, *reuse* of previously learnt lemmas from overlapping BMC instances to improve SMT solver performance.
6. We transform the model (preserving LTL\X property) using *COI reduction*, *Collapsing*, and *Balancing Paths and Loops*, so as to improve the scope of learning and simplification based on high-level information.

3. Preliminaries

3.1. Extended Finite State Machine (EFSM) Model

Our method extracts high-level information from an Extended Finite State Machine (EFSM) model of a sequential program/design, with a partitioning of control and datapath. An EFSM has finite logical (control) states and conditionals (guards) on the transitions between the control states. The guards are functions of control states, data-path and input variables. Formally, an EFSM model is a 6-tuple $<s_0,S,I,O,D,T>$ where, s_0 is an initial state, S is a set of control states (or blocks), I is a set of inputs, O is a set of outputs, D is a set of state (datapath) variables, $T=(T_E, T_U)$ is a transition relation, with T_E being an enabling transition relation, T_E: $S\times D\times I\rightarrow S$, and T_U being an update transition relation, T_U: $S\times D\times I\rightarrow D\times O$.

An ordered pair $<s,x> \in S\times D$ is called a *configuration* of M. A transition from a configuration $<s,x>$ to $<t,y>$ under an input i, with possible output o comprises of two transitions: a) an enabling transition, represented as $((s,x,i),(t))\in T_E$, and b) an update transition, represented as $((s,x,i),(y,o))\in T_U$. For a given enabling transition $s\rightarrow t$, we define an enabling function f such that $f(x,i)=1$ *iff* $((s,x,i),(t))\in T_E$ and we label the transition as $s\rightarrow_{f(x,i)} t$. For ease of description, we consider deterministic EFSMs where for any two transitions from a state s, *i.e.* $s\rightarrow_{f(x,i)} t_1$ and $s\rightarrow_{g(x,i)} t_2$, $f(x,i) \wedge g(x,i) = FALSE$. We define $to(s) = \{t \mid s\rightarrow_{f(x,i)} t\}$ as a set of outgoing control states of s. Similarly, we define $from(t) = \{s \mid (s,t)\in T_E\}$ as a set of incoming control states of t. We define a NOP state as a control state with no update transition and a single outgoing enabling transition. A NOP state with n incoming transitions can be replaced with n NOP states, each with a single incoming and a single outgoing transition, without changing incoming or outgoing states. In our discussion, a NOP state will have only a single incoming/outgoing transition. We define a SINK state as a control state with no update transition relation and no outgoing transition. We define a transition or state as *contributing* with respect to a variable if it can affect the variable; otherwise, such a transition or state is called *non-contributing*.

Example 1: We illustrate EFSM model M of a FIFO example (implemented using a RAM) using a State Transition Graph (STG) as shown in Figure 2(a) where *S={S0,…,S11}*, *I={ren, wen, fifo_in}*, *O={fifo_out, is_full, is_empty}*, *D={rptr, wptr, is_full, is_empty, RAM[10]}*. The enabling functions are shown in *italics* and update transitions are shown in non-italics in the Figure 2(a). For example, the transition $S2\rightarrow_{T23}S3$ has enabling function *T23=(rptr==wptr-1)||(rptr=0 && wptr==9))* and update transition *{RAM[rptr]=in; is_empty=0}*.

3.2. Flow Graphs

A *flow graph* $G(V,E,r)$ is a directed graph with an entry node r. A path from u to v, denoted as $p(u,v)$, is a sequence of nodes $u=s_1,\ldots,s_k=v$ such that $(s_i,s_{i+1})\in E$ for $1\le i<k$. We denote $Path(u,v)$ as a set of paths between u and v. Length of a path $p\in Path(u,v)$ is the number of edges in the sequence. For $p_x,p_y\in Path(u,v)$, $p_x\neq p_y$ if the sequence of states in p_x is different from p_y; such paths are called *re-converging* paths. A concatenation of paths $p(t,u)$ and $p(u,v)$, denoted as $p(t,v)=p(t,u)\oplus p(u,v)$, represents a path from t to v that goes through u. A node n is said to *dominate* node m, denoted as $dom(m)$ if every path from r to m goes through n. The node r dominates every other node in the graph. A Strongly Connected Component, SCC (V_i,E_i) is a subgraph of G such that for all $u,v\in V_i$, there exists a path $u=s_1,\ldots,s_k=v$ such that $(s_j,s_{j+1})\in E_i$ for $1\le j<k$. SCC (V_i,E_i,r_i) is a loop with entry r_i if r_i dominates all the nodes in V_i. An edge (u,v) is called a *back-edge* if v dominates u; otherwise it is called a *forward-edge*. Given a back-edge (u,v), a *natural loop* of the edge is defined as the set of nodes (including u) that can reach u without going through v. For a given $G(V,E,r)$, and any pair of natural loops, $L_1= (V_1,E_1,r_1)$ and $L_2= (V_2,E_2,r_2)$ one of the following cases holds: 1) they are disjoint i.e., $V_1\cap V_2=\varnothing$, 2) they are disjoint but have the same entry node i.e. $V_1\cap V_2=\{r_1\}=\{r_2\}$, or 3) one is completely nested in the other, i.e. $V_1\subseteq V_2$ *or* $V_2\subseteq V_1$. A *sink* node is a node with no outgoing edge.

A flow graph $G(V,E,r)$ is *reducible* [19] if and only if E can be partitioned into disjoints sets *front-edge set* E^f and *back-edge set* E^b such that $G^d(V,E^f,r)$ forms a direct-acyclic graph (DAG) where each $v\in V$ can be reached from the entry node r. The reducible graph has the property that there is no jump into the middle of the loops from outside and there is only one entry node per loop. Most flow graphs that occur frequently fall into the class of reducible flow graphs. Use of structured control flow statements such as *if-then-else, while-do, continue* and *break* produces programs whose flow graphs are always reducible. Unstructured programs due to the use of *goto* can cause irreducibility of the graphs. Thus focusing on a reducible

graph is not a significant restriction of our algorithms and techniques, and indeed accords with practical guidelines.

For a given EFSM $<s_0,S,I,O,D,T>$, let $G=(V,E,r)$ be a flow graph with start vertex r, such that $V=S$, $E=\{(s,t)|\ s \rightarrow_t\}$, and $r = s_0$. The sets $to(s)$ and $from(s)$ represent the set of *outgoing* nodes and *incoming* nodes of a node s, respectively. A reachability analysis on a flow graph corresponds to control state reachability analysis of the corresponding EFSM.

4. Our Approach: High-level BMC

We present the flow of our approach for high-level BMC as shown in Figure 1. Given an EFSM Model M (discussed in 3.1) and a property P, we perform a series of *novel* property preserving transformations (Sections 4.3 and 4.6). After that we perform control state reachability on the transformed model (Section 4.4). Using the reachability information, we generate *novel* simplification constraints on-the-fly at each unroll depth k (Section 4.5). These simplification constraints are used by the expression simplifier (Section 4.1) during unrolling to reduce the formula. These constraints are also used to improve the search on the translated problem. We also propose an incremental learning technique (Section 4.2) i.e., re-use of theory lemmas in high-level BMC framework. We present various innovations in the order of ease of explanation.

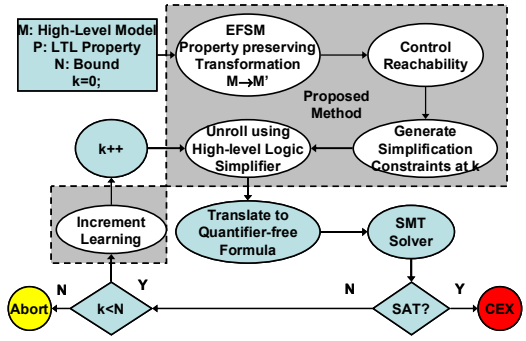

Fig. 1: Accelerated High-level BMC

4.1. 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 ($<,>,\geq,\leq,==$) between term expressions, and *uninterpreted predicate*s (*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 behaviour 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 re-writes expressions using local and recursive transformations in order to remove structural and multi-level functionally redundant expressions, similar to simplifications proposed for Boolean logic [6, 20] and also for first order logic [21].. Our expression simplifier has a "compose" operator [7], 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.

4.2. Incremental Learning in High-level BMC

Learning from overlapping instances of propositional formulas has been proposed previously [3, 7, 8] and found to be useful in Boolean SAT-based BMC [3, 4, 22]. We use incremental learning of theory lemmas across time-frames, and found this technique to be equally beneficial in the context of high-level BMC.

4.3. Property-based EFSM Reduction

We perform slicing on EFSM [23] with respect to variables of interest as defined by the property and obtain *contributing* and *non-contributing* states and transitions. Slicing away behaviors (and the elements) unrelated to the specific properties can significantly reduce the model size and thereby, improve the verification efforts. We describe two such techniques in the following: *cone-of-influence (COI) reduction* and *collapsing*.

4.3.1. COI reduction

- We remove all non-contributing states and their outgoing transitions.
- Any non-contributing transition $s \rightarrow_{f(x,i)} t$ where s is a contributing state, is replaced by a transition $s \rightarrow_{f(x,i)} SINK$.
- If we are concerned with reachability of a state $s \in S$ from a start state s_0, we remove the outgoing transition from s since it is non-contributing for the shortest counter-example or proof. For example, the self-loop transition $S1 \rightarrow S1$ (not shown in Figure 2(a)) is *non-contributing* and hence is replaced by $S1 \rightarrow SINK$ as shown in Figure 2(a).

4.3.2. Collapsing

We define a *collapsing condition* as that when all states in $to(s)$ are NOP and none of them directly appears in a reachability check. Under such a condition, we collapse all the NOP states and merge them with s. In other words, $\forall\, t \in to(s)$ (with t being NOP), we remove the transitions $s \rightarrow_{f(x,i)} t$ and $t \rightarrow_{TRUE} q$ and add a new transition $s \rightarrow_{f(x,i)} q$.

4.4. Extraction: Control State Reachability (CSR)

We now discuss extraction of high-level control-flow information of the design/program which is subsequently used to simplify the unrolled formula (discussed in the next section).

Starting from the initial state $S0$, we compute control state reachability (CSR) using a breadth first search (BFS). A control state S_i is one-step reachable from S_j *if and only if* there exists an enabling transition between them. At a given sequential depth d, let $R(d)$ represent the set of control states that can be *reached statically* in one step from the states in $R(d-1)$ with $R(0)=\{S0\}$. Note that we are not computing the fixed-point diameter. For some d, if $R(d-1) \neq R(d)=R(d+1)$, we say the CSR *saturates* at depth d and stop; otherwise we compute $R(d)$ and $|R(d)|$ (i.e., size of $R(d)$) up to the BMC bound. Note, CSR information is static information without considering the enabling functions, i.e., if a control state is not reachable from the initial state in CSR, it is definitely not reachable in the model; however, the other way is not true in general. Applying CSR on the FIFO example 1, we obtain the set $R(d)$ as shown in Table 1(a). Note,

the saturation depth is 15, $|R(15)|=|R(16)|=11$ where $R(15)=\{S1,S2,...,S11\}$.

4.5. On-the-Fly Simplification and Learning

For n control states $S1...Sn$, we introduce n Boolean variables $B_{S1}...B_{Sn}$. Let the Boolean variable $B_r = TRUE$ iff configuration of M is (r,x) for some $x \in V$. Equivalently, B_r corresponds to a predicate on the control state variable, called the *PC (Program Counter)*, i.e., $B_r \equiv (PC==r)$. Let B_r^d denote the Boolean variable B_r at depth d during BMC unrolling.

At any unrolling depth d of high-level BMC, we apply the following on-the-fly structural and clausal (learning-based) simplification on the corresponding formula. Note, these simplifications are effective for small $|R(d)|$. We use a procedure *Simplify (BoolExpr e, Boolean v)* which constraints a Boolean expression e to a Boolean value v, and also reduces the expressions that use e. Later, we illustrate this with an example.

1. Unreachable Block Constraint (UBC)

$\forall_{r \notin R(d)}$ $Simplify(B_r^d, 0)$

Since the state r is not reachable at depth d, the predicate B_r will evaluate to *FALSE* at depth d. Therefore, simplifying the formula by propagating $B_r=0$ at depth d preserves the behavior of the design.

2. Reachable Block Constraint (RBC)

$Simplify(\vee_{r \in R(d)} B_r^d, 1)$

At any depth d, at least one state in $R(d)$ is reachable.

3. Mutual Exclusion Constraint (MEC)

$\forall_{r,t \in R(d),\ r \neq t}$ $Simplify((B_r^d \Rightarrow \neg B_t^d), 1)$

At any depth d, at most one state in $R(d)$ is the current state.

4. Forward Reachable Block Constraint (FRBC)

$\forall_{r \in R(d)}$ $Simplify((B_r^d \Rightarrow \vee_{s \in to(r)} B_s^{d+1}), 1)$

At any depth d, if current state is r i.e. $B_r^d=TRUE$, then the next state must be among the $to(r)$ set.

5. Backward Reachable Block Constraint (BRBC)

$\forall_{r \in R(d)}$ $Simplify((B_r^d \Rightarrow \vee_{s \in from(r)} B_s^{d-1}), 1)$

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 $from(r)$ set.

6. Block-Specific Invariant (BSI)

$\forall_{r \in R(d)}$ $Simplify((B_r^d \Rightarrow C_r^d), 1)$

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.

Note, previous approaches [24] add some of these constraints in the transition relation so as to include them in the formula at every unrolling. In contrast, our approach adds only the *relevant* constraints at each unrolling, thereby reducing the overall formula size. Thus, ideally we would like a smaller set $R(d)$ to increase the effectiveness of our simplification. Later, we discuss how we transform EFSM model to reduce the set $R(d)$.

Example 1(Contd): We illustrate simplification constraints at depth, $d=4$. In particular, we consider the effect of simplification on the unrolled expression for variable *is_full*. The transition relation for the state variable *is_full* is as follows:

$next(is\_full) = (B_{S0} \,||\, B_{S7})? \ 0 : (B_{S3}) \ ? \ 1 : is\_full;$

The high-level expression for the unrolled variable, corresponding to *next(is_full)* at depth 5, would be:

$is\_full^5 = (B_{S0}^4 \,||\, B_{S7}^4)? \ 0 : (B_{S3}^4) \ ? \ 1 : is\_full^4;$

For lack of space, we explain only the *Unreachable Block Constraint*. Note, at $d=4$, only *S4, S5, S6, S9, S10,* and *S11* are reachable (Table 1(a)). Therefore, we do the following:

$\forall_{r \in \{S0,S1,S2,S3,S7,S8\}}$ $Simplify(B_r^4, 0)$

Using the above simplification, the expression for is_full^5 gets mapped to an existing variable is_full^4, thereby, reducing the additional logic, i.e., $is\_full^5 = is\_full^4$.

4.6 EFSM Transformation: Balancing Re-convergence

Efficiency of on-the-fly simplification depends on the size of the set $R(d)$, *i.e.,* $|R(d)|$. A larger $|R(d)|$ reduces the scope of simplification at depth d and hence, the performance of high-level BMC. Re-converging paths of different lengths inside loops is one of the reasons for the saturation of reachability and inclusion of all looping control states in the set R. To improve the performance of high-level BMC further, we adopt a strategy called "Balancing Re-convergence" that transforms the original model into a "BMC friendly" model but preserves the validity of the model with respect to the property expressed in LTL\X (Linear Temporal Logic without the neXt-time operator).

4.6.1. Our Strategy: Intuition

For balancing re-convergence and reducing the set $R(d)$ and thereby, improving the scope of simplification of high-level BMC, *we transform an EFSM by inserting NOP states such that lengths of the re-convergence paths are the same and control state reachability does not saturate*. Reduction in $R(d)$, in general, improves the scope of on-the-fly simplifications. Note that the additional NOP states have little effect on simplification, although they increase the total number of control states and transitions, and possibly the search depth. As NOP states do not add complexity to the transition relations of any state variables except the program variables encoding the control states, the *Unreachable Block Constraint* simplification at depth d is practically unaffected by inclusion of such states in $R(d)$. Also, the *Forward* and *Backward Block Constraint* simplification are not affected, as these additional transitions are single outgoing transition (and hence always enabled) from NOP states. We define $R^-(d) = \{s \mid s \in R(d) \text{ and } s \text{ is not a NOP state}\}$. We use $max_d\ |R(d)|$ and $max_d\ |R^-(d)|$ to measure the effectiveness of our strategy in improving the scope of simplification of high-level BMC. Note that the above transformation preserves LTL\X properties, as NOP states can only increase the length of traces but not the eventuality and global behavior. As the state of data variables do not change in NOP state, the validity of atomic propositions is not affected.

Example 1 (Contd): For the EFSM model shown in the Figure 2(a), paths $S2 \rightarrow S3 \rightarrow S4$ and $S2 \rightarrow S4$ are re-converging with different lengths. For balancing, we insert a NOP state *S2a* such that transition $S2 \rightarrow_{\sim T23} S4$ is replaced by $S2 \rightarrow_{\sim T23} S2a \rightarrow_{TRUE} S4$. Similarly, as *paths* $S7 \rightarrow S8 \rightarrow S9$ and $S7 \rightarrow S9$ are re-converging with different lengths, we insert another NOP state *S7a* and replace the transition $S7 \rightarrow_{\sim T78} S9$ by $S7 \rightarrow_{\sim T78} S7a \rightarrow_{TRUE} S9$. The modified EFSM model M' is shown in the Figure 2(b). CSR on *M'* is shown in Table 1(b). Note that at depth $max_d\ R^-(d) = 4$. Also, CSR on *M'* does not saturate.

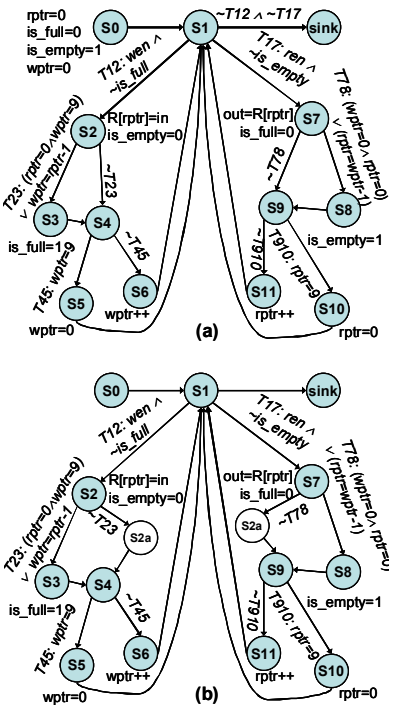

Fig. 2: STG of EFSM a) original M b) transformed M'

Table 1[a-b]: Control State Reachability on EFSM a) M b) M'

(a) Model M			(b) Model M'			
d	**R(d)**	**|R(d)|**	**d**	**R(d)**	**|R(d)|**	**|R⁻(d)|**
0	S0	1	0	S0	1	1
1	S1	1	1	S1	1	1
2	S2, S7	2	2	S2, S7	2	2
3	S3, S4, S8, S9	4	3	S3, S2a, S8, S7a	4	2
4	S4, S5, S6, S9, S10,S11	6	4	S4, S9	2	2
5	S5, S6,S10, S11, S1	5	5	S5, S6, S10, S11	4	4
..	…..		6	S1	1	1
15	Saturates with 11 states		i	Repeats, R(i)=R(i%5)		

Algorithm: Given a *reducible* flow graph G, we present an *O(E)* algorithm in Sections 4.6.2 and 4.6.3, that identify the edges corresponding to the transitions in T_E where inserting certain number of *NOP* states will balance the re-convergence paths, including those arising due to loops.

4.6.2. Balancing Re-convergence without Loops

Consider the DAG, $G(V,E^f,r)$ corresponding to the reducible flow graph $G(V,E,r)$ with an entry node r and front-edge set E^f. Let $w(e)$ denote the weight of the edge, $e=(a,b)\in E^f$. As we later see, the weight of the edge (a,b) corresponds to one more than the number of NOP states that need to be inserted between nodes a and b. We define weight for a path $p=\langle s_1,\ldots,s_k\rangle$, denoted as $w(p) = \Sigma_{1\le i<k}\, w(e_i)$ where $e_i=(s_i,s_{i+1})\in E^f$. We now define our problem as follows:

Problem 1: For a given DAG $G(V,E^f,r)$ find a weight function, $w:E^f\rightarrow Z$ such that $\forall p_x,p_y\in P(u,v)\ w(p_x)=w(p_y)$, where $u,v\in V$.

Note, if we are able to find a feasible w, then the number of NOP states introduced for an edge, $e=(a,b)\in E^f$ will be equal to $w(e)-1$. We say that the set $P(u,v)$ is *balanced* when all the paths from u to v have equal weights, i.e., $\forall p_x,p_y\in P(u,v),\ w(p_x)=w(p_y)$.

Let $W(u)$ denote the weight of the paths in the balanced set $P(r,u)$. We define $W(r)=0$.

Lemma 1: If $P(r,v)$ is balanced and $P(u,v)\neq\emptyset$, then $P(u,v)$ is balanced.

Proof: We prove by contradiction. Let $p_1(r,u)\in P(r,u)$. Assume $P(u,v)$ is not balanced, i.e., there exists at least two paths $p_1(u,v)\in P(u,v)$ and $p_2(u,v)\in P(u,v)$ such that $w(p_1)\neq w(p_2)$. Let $p_0(r,u)\in P(r,u)$. The weight of the concatenated path $p_0\oplus p_1$ is $w(p_0)+w(p_1)$ and that of $p_0\oplus p_2$ is $w(p_0)+w(p_2)$. Since $w(p_1)\neq w(p_2)$, the weight of the concatenated paths are different. However, since $p_0\oplus p_1$, $p_0\oplus p_2\in P(r,v)$ and $P(r,v)$ is balanced, we get a contradiction. Thus, $P(u,v)$ is also balanced.

Using *Lemma 1*, we re-formulate the problem as follows:

Problem 1' (stated differently): Given a DAG $G(V,E^f,r)$, find a weight function, $w:E^f\rightarrow Z$ and $W:V\rightarrow Z$ such that $P(r,v)$ is balanced i.e., $w(p_x)=w(p_y)=W(v),\ \forall p_x,p_y\in P(r,v)$

Solution: If $P(r,u)$ is balanced $\forall u\in from(v)$, *i.e.,* $W(u)$ is computed, we can balance $P(r,v)$ *recursively* as follows.

- $\forall_{u\in from(v)}\ w(u,v) = t-W(u)$, where $t = (\max_{u\in from(v)} W(u))+1$
- Set $W(v)=t$ as for any path $p(r,v)$ through u will have weight $W(u)+w(u,v)=W(u)+t-W(u)=t$.

We start with an initial set of nodes S which are *sink nodes* in $G(V,E^f,r)$. Then, we recursively apply the above steps in the procedure *BalancePath,* as shown in Figure 3. Termination is guaranteed as the recursive sub-procedure *BalanceAux* is invoked only once per node. The correctness of the algorithm is also shown easily by an inductive argument.

```
1.  Input: G(V,E^f,r)
2.  Output: w:E->Z, W:V->Z
3.  Procedure: BalancePath
4.  S = {v | v is a sink node}
5.  W(r) = 0; ∀v∈V,v≠r W(v)=∞;
6.  ∀v∈S, BalanceAux(v);

7.  Input: v
8.  Output: W(v)
9.  Proecedure: BalanceAux
10. ∀u∈from(v)
11.  if (W(u)==∞) BalanceAux(u);
12. W(v) = Max u∈from(v) (W(u))+1;
13. ∀u∈from(v), w(u,v)=W(v)-W(u);
```

Fig. 3: Pseudo-code of *BalancePath*

Collapsing NOP states

As discussed earlier, we insert NOP states corresponding to the edge weights obtained after running the procedure *BalancePath.* For edge $e=(u,v)$, we insert $(w(e)-1)$ NOP states. It is easy to see that the algorithm *BalancePath* adds a minimum number of NOP states for balancing paths. However, the inserted NOP states together with NOP states in the original EFSM can generate a *collapsing condition* (discussed in Section 4.3.2). In that case, we collapse the NOP states as discussed earlier. We re-run *BalancePath* as the lengths of the re-converging paths might have changed due to collapsing. Note, we can integrate collapsing with the procedure *BalancePath* to avoid re-running.

4.6.3. Balancing Re-convergence with Loops

Since the flow graph $G(V,E,r)$ is reducible, we know that every loop $L_i=G(V_i,E_i,r_i)$ is a natural loop corresponding to backedge set $(b_i,r_i)\in E^b$, and has a single entry node r_i. Presence of back-edges in loops and their relative skews cause re-convergence

paths of different lengths; which in turn, can also lead to saturation during control reachability analysis. We say a loop L_i is saturated at depth s when $\forall v \in V_i$, $v \in R(t)$ for $t \geq s$. Given balanced $Path(r_i,b_i)$ for each loop L_i, we define *forward loop length*, C_i for loop L_i as follows:

$$C_i = W(b_i) - W(r_i)$$

where $W: V \rightarrow Z$ is the weight function we obtain for each node in $G(V,E^f,r)$ using the *BalancePath* algorithm, shown in Figure 3. Observe that the entry node r_i of loop L_i re-appears in control reachability after $N_i = C_i + w(b_i,r_i)$ steps i.e. $r_i \in R(d_i + n_i N_i)$ where $d_i, n_i \in Z$. We call N_i the *loop period* of L_i. If there is only one loop, it is easy to see that $d_i = W(r_i)$. However, in presence of multiple loops, we also have to account for the paths from other loops to loop L_i. In particular, if there is a path from entry node r_j of some loop L_j to r_i, then entry r_i also re-appears after N_j. We define *loop clusters LC* as sets of disjoint entry nodes such that for any two clusters LC_x and LC_y, $\forall s \in LC_x$, $\forall t \in LC_y$, $P(s,t) = P(t,s) = \Phi$. Note, a loop in a cluster does not affect the loop in another cluster as far as reachability is concerned. In the following problem statement, we discuss how to prevent loop saturation using suitable transformations.

Problem 2: Given a reducible flow graph $G(V,E,r)$ with $E = E^f \cup E^b$, find $w: E^b \rightarrow Z$ and N_i so that loop L_i is not saturated.

Solution: We define a set $from(i) = \{j \mid r_j = r_i \text{ or } Path(r_j,r_i) \neq \Phi\}$. Thus, $d_i = W(r_i) + \Sigma_{j \in from(i)} n_j N_j$ where $n_j \in Z$. Define, $D_i = min_{k \in from(i) \cup \{i\}} N_k$. It is easy to see that a loop L_i gets saturated at depth $t + D_i$ during reachability if $r_i \in R(t+k)$ $\forall k$, $0 \leq k < D_i$. This is captured by the following integer linear equations in terms of s and n_j's for given N_j's, N_i and $W(r_i)$.

$$W(r_i) + \Sigma_{j \in from(i)} n_j N_j + n_i N_i = s$$
$$W(r_i) + \Sigma_{j \in from(i)} n_j N_j + n_i N_i = s+1$$
$$\ldots$$
$$W(r_i) + \Sigma_{j \in from(i)} n_j N_j + n_i N_i = s + D_i - 1$$

To prevent saturation of loop L_i, we need to find N_j's and N_i such that there is no feasible solution to the above equations. One strategy is to choose a weight function $w: E^b \rightarrow Z$ such that the loop lengths match i.e., $N_i = N_j$ $\forall j \in from(i)$. (It is easy to verify the infeasibility for this solution assuming that each loop has at least two nodes, *i.e.*, $N_i \geq 2$.)

We consider one *loop cluster* at a time. We define *maximum loop period* over all loops in the cluster (i.e. whose entry nodes are in the cluster), $N = (Max_i\ C_i) + 1$. We assign a weight to each back-edge (b_i,r_i) as follows:

$$w(b_i,r_i) = N - C_i$$

For each loop L_i in the cluster, the entry node $r_i \in R(W(r_i) + nN)$ where $n \in Z$. Thus, the upper bound on $|R(d)|$ for $G(V,E,r)$ at any depth $d >> 1$, $|R(d)| \leq \Sigma_i\ max_t\ |R_i(t)|$, where $R_i(t)$ is the control reachability set (including NOP states) on loop L_i at a depth t. Similarly, the upper bound on $|R^-(d)|$ for $G(V,E,r)$ at any depth $d >> 1$, is $|R^-(d)| \leq \Sigma_i\ max_t\ |R_i^-(t)|$, where $R_i^-(t)$ is the control reachability set (of only non-NOP states) on loop L_i at a depth t.

Example 2: We illustrate our algorithm for balancing flow graph using an example shown in Figure 4(a). Let v_i represent the node with index i (shown inside the circle). Note, the flow graph $G(V,E,v_1)$ has three natural loops L_1, L_2 and L_3 corresponding to the back-edges (v_6,v_3), (v_7,v_1), and (v_8,v_3) respectively. The corresponding entry nodes for the loops are v_3, v_1, and v_3 respectively. Note, they all form a *cluster*. The DAG $G(V,E^f,v_1)$ corresponding to the front-edge set, $E^f = E - \{(v_6,v_3), (v_7,v_1), (v_8,v_3)\}$ is shown in Figure 4(b). After executing *BalancePath* algorithm, we obtain edge weights, also shown in Figure 4(b), that balance all re-convergence paths in E^f. Note that the edge with no weight shown has an implicit weight of 1. Also, shown are the W values of each node. For instance, $W(v_6) = 5$ denotes that all the paths in the set $P(v_1,v_6)$ have weights equal to 5. Next, we compute the *forward loop length* of each loop and the weights of the back-edges. The *forward loop length* of loop with back-edge (v_6,v_3) is $W(v_6) - W(v_3) = 3$; similarly, with back-edge (v_7,v_1) is 6, and with back-edge (v_8,v_3) is 5. Thus, value of N, as defined, is 7. The weight of the back-edges (v_6,v_3), (v_7,v_1) and (v_8,v_3) are 4, 1, and 2 respectively as shown in Figure 4(c). For each edge with weight w, we insert $w-1$ nodes corresponding to NOP states as shown as un-shaded circles in the modified flow graph in Figure 4(d).

Reachability on the original flow graph $G(V,E,v_1)$ in Figure 4(a) saturates at depth 6 with 8 nodes. The reachability on the balanced flow graph in Figure 4(d) does not saturate. Instead, the set of reachable nodes $R(d)$ at depth d shows a periodic behavior with period, N=7. If we do reachability separately on each loop of the modified flow graph in Figure 4(d), we obtain $max_t\ |R_1(t)| = 2$, $max_t |R_2(t)| = 2$, and $max_t |R_3(t)| = 2$. Thus, the upper bound on $|R(t)|$ is 6. Similarly, $max_t\ |R^-_1(t)| = 1$, $max_t |R^-_2(t)| = 1$ and $max_t |R^-_3(t)| = 1$ and upper bound on $|R^-(t)|$ is 3. In this case, $max_t |R(t)| = 4$ and $max_t |R^-(t)| = 2$. Clearly, the scope of simplification during BMC is significantly improved.

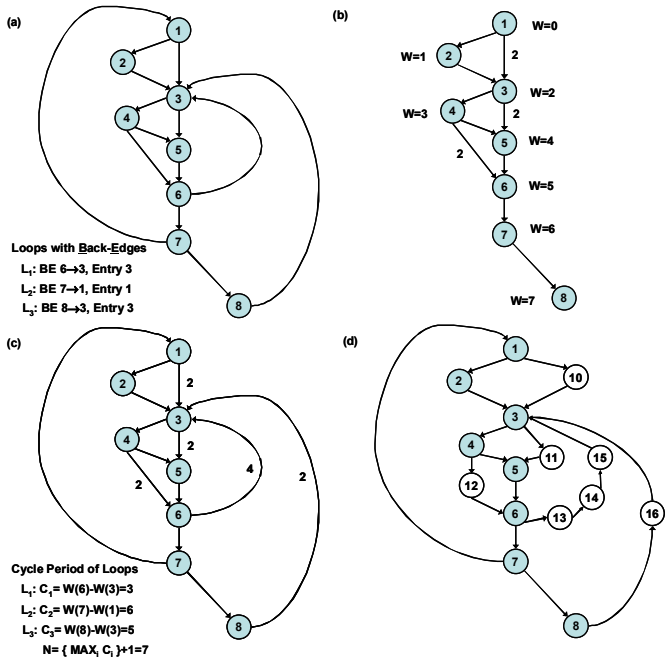

Fig. 4: Execution steps of Balancing Re-convergence on an example: a) Reducible Flow graph $G(V,E,v_1)$ where i represents the node v_i, b) DAG $G(V,E^f,v_1)$ with edge weights (=1 if not shown) after executing *BalancePath* procedure, c) weights on the back-edges after balancing loops, d) final balanced flow graph after inserting $n-1$ NOP states for edge with weight n.

5. Experiments

We experimented on a public benchmark *bc-1.06*, a C program for an arbitrary precision calculator language with interactive execution of statements. This has a known *array bound access* bug (checked as an error-label reachability property). Using our

program verification tool F-soft [24], we first generated an EFSM model *M* with 36 control states and 24 state variables. The data path elements include 10 *adders*, 106 *if-then-else*, 52 *constant multipliers*, 11 *inequalities*, and 49 *equalities*. The corresponding flow graph has two loops, with 4 and 8 nodes (control states) respectively. We also used statically generated invariants [25] to provide block specific invariants.

We performed controlled experiments to evaluate the role of various accelerators discussed in improving the performance of high-level BMC. We used our *difference logic solver SLICE* [14] in the backend. We modified the solver to support incremental learning across time-frames. We translated conservatively each BMC problem instance into a difference logic problem. (A precise translation would have been to a UTVPI – Unit Two Variables Per Inequality – problem.) For understanding the effectiveness of our methods, a conservative translation suffices as long as we do not get false negatives (which was not an issue for this example).

We conducted our experiments on a workstation with dual Intel 2.8 GHz Xeon Processors with 4GB physical memory running Red Hat Linux 7.2, using a 500s time limit for each high-level BMC run. We present the results in Table 2. We experimented on three EFSM models *M, M' and M''*. Model *M* is the original model without any proposed transformations. Model *M'* is the model obtained from *M* using the procedure described in Section 4.6.2 (*Balancing re-convergence without loops* and *Collapsing NOP states*). Model *M''* is obtained from *M'* using the procedure described in Section 4.6.3 (*Balancing re-convergence with loops*). Column 1 shows the loop sizes in each of the models for loops L_1 and L_2; the number of control states (including inserted NOP states); the results of control reachability on each of the models i.e., either saturation depth or max loop period *N*; maximum size of the reachable set of control states overall all depth $R_{max}=max_d R(d)$; and maximum size of the reachable set of *non-NOP* control states overall depth, $R^-_{max}=max_d R^-(d)$. Column 2 presents various learning and simplification strategies denoted as follows: *A* for *Expression Simplification (ES)*, *B* for *Incremental Learning (IL)* combined with *A*, *C* for *Unreachable Block Constraint* (*UBC*) combined with *B*, *D* for *Reachable Block Constraint (RBC)* combined with *C*, *E* for *Forward Reachable Block Constraint (FRBC)* combined with *D*, *F* for *Backward Reachable Block Constraint (BRBC)* combined with *E*, and *G* for *Block Specific Invariants (BSI)* combined with *F*. Column 3 shows number of calls (*#HS*) made to the high-level solver when the expression simplifier cannot reduce the problem to a tautology. Column 4 shows the depth *D* reached by high-level BMC under a given time limit ('*' denote time-out). Column 5 shows the time taken (in seconds) to find the witness; *TO* denotes that time-out occurred. Column 6 shows whether a witness was found in the given time limit; if so, the witness length is equal to D.

5.1. Discussion of results

Note that fewer calls (#HS) made to the SMT solver directly translates into performance improvement, as the expression simplifier structurally solves the remaining D-#HS SMT problems more efficiently. We discuss the effect of various learning scheme in improving the structural simplifications. CSR on Model M saturates at depth 13 with 36 control states. Although *Unreachable Block Constraint (UBC)* allows deeper search with fewer solver calls, the simplification scope is very limited due to a large set *R(d)*. This also prevents other simplification strategies from being useful. As shown in Column 6, none of the strategies is able to find the witness in the given time limit. When we apply the procedure *BalancePath* with the procedure *collapsing of NOP states* on Model *M*, we obtain a model with *M'* with 34 control states with reduced loop size $|L_2|$. CSR on *M'* does not saturate, and has $max_d R(d)=4$ and $max_d R^-(d)=3$. This increases the scope of simplification significantly. As shown in Column 6, all simplification strategies C-G are able to find the witness in the given time limit. Except for *FRBC*, all simplification strategies seem useful in reducing the search time; though only *UBC* can reduce the number of calls to the high-level solver as shown in Column 3. *Block Specific Invariants* added on-the-fly are also found to be useful. Note, although strategy *B* with only incremental learning does not find the witness, it still helps to search deeper compared to strategy *A*.

Table 2: Comparison of high-level BMC accelerators

Model	Strategy	#HS	D	sec	W?
Original M $\|L_1\|$= 4, $\|L_2\|$=8, #ctrl state=36 R_{max}=36, R^-_{max}=33 Saturation at d=13	A: ES	16	17*	TO	N
	B: A+ IL	26	27*	TO	N
	C:B + UBC	41	64*	TO	N
	D:C + RBC	26	49*	TO	N
	E:D+FRBC	26	49*	TO	N
	F:E+BRBC	28	51*	TO	N
	G: F + BSI	28	51*	TO	N
M': M+Balanced Non-Loop paths + collapsed NOP states $\|L_1\|$= 4, $\|L_2\|$=6, #ctrl state=34 R_{max}=4, R^-_{max}=3, Max loop period, N=6	B	28	29*	TO	N
	C	62	143	426	Y
	D	62	143	159	Y
	E	62	143	159	Y
	F	62	143	120	Y
	G	62	143	65	Y
M'':M'+Loop Balanced $\|L_1\|$= 6, $\|L_2\|$=6, #ctrl state=36 R_{max}=3, R^-_{max}(d)=2, N=6	F	32	205	19	Y
	G	32	205	22	Y

By applying our loop balancing procedure on the model *M'*, we obtain a model *M''* with matching loop lengths of 6 and total number of control states of 36. We added two NOP-states in the back-edge of loop L_1 to get a loop length of 6. Control reachability on *M''* has $max_d R(d)=3$ and $max_d R^-(d)=2$, further increasing the scope of simplification as indicated by a decreased number of calls to the high-level solver. This is indicated by the reduced solve time (=19s) using strategy *F*, although there is a small performance degradation with strategy *G*. Not surprisingly, the witness length has gone up to 205. Overall, we see progressive and cumulative improvements with various learning techniques and strategies.

5.2. Comparison with Boolean-level BMC

To compare with Boolean-level BMC, we used our state-of-the-art Boolean-level BMC framework *DiVer* [4] on a Boolean translation of the model M (with 654 latches, 6K gates) to witness the bug, and used an identical experimental setup as discussed. Note, like in [24], we add high-level information such as *mutual exclusion constraint* and *backward reachable block constraints* in the transition relation beforehand. Thus, all these constraints get included in every unrolled BMC instance automatically, unlike the proposed approach here, where only the relevant constraints are added to a BMC instance. The Boolean-level BMC is able to find a witness at depth 143 in 723s. Not surprisingly, the number of instances solved by structural simplification is merely 15, while 128 calls are made

to the SAT-solver. Thus, a reduced scope of simplification can greatly affect the performance of BMC, further supporting the case for synthesizing "BMC friendly" models [26].

6. Experiments on Industry Software

We also 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 *un-interpreted 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. Using a similar experimental setup (discussed earlier), we ran high-level BMC (HBMC) for 500s on each of P1-P6 on: (I) Model *M* with strategy *A* (using only expression simplification), (II) Model *M* using strategy *F* (all simplifications), and (III) transformed Model M" using *F*. We present our results in Table 3. Column 1 gives the property checked; Column 2-4 give 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.

Table 3: Evaluating high-level BMC on industry software

P	I: Strategy A on M			II: Strategy F on M			III: Strategy F on M''		
	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

7. Conclusions and Future Work

The current trend of designing at higher levels of abstraction using high-level languages and specifications has challenged the verification community to lift the maturity and advancements of BMC from the Boolean-level to the higher levels. Although high-level BMC overcomes several inherent limitations of Boolean-level BMC, higher theoretical complexity of the associated logics and decision procedures makes the approach even more challenging. We provide an engineering framework for high-level BMC with several state-of-the-art innovations based on extraction and efficient use of high-level information to improve the performance and scalability. This framework also allows easy integrations of the state-of-the-art techniques available for Boolean-level BMC. We believe that our proposed framework is a step towards reducing the gap between theory and practice of such techniques.

References

[1] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proceedings of the DAC*, 1999.

[2] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties using Induction and a SAT Solver," in *Proceedings of Conference on FMCAD*, 2000.

[3] O. Strichman, "Pruning Techniques for the SAT-based bounded model checking," in *Proceedings of TACAS*, 2001.

[4] M. Ganai, A. Gupta, and P. Ashar, "DiVer: SAT-Based Model Checking Platform for Verifying Large Scale Systems," in *Proceeding of TACAS*, 2005.

[5] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," in *Proceeding of CAV*, 2002.

[6] H. Andersen and H. Hulgard, "Boolean expression diagram," in *Proceedings of LICS*, 1997.

[7] M. Ganai and A. Aziz, "Improved SAT-based Bounded Reachability Analysis," in *Proceedings of VLSI Design Conference*, 2002.

[8] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," *Proceedings of DAC*, 2001.

[9] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*: Kluwer Academic Publishers, 1993.

[10] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, M. Schulz, and R. Sebastiani, "The MathSAT 3 System," in *Proceedings of CADE*, 2005.

[11] C. Barrett, D. L. Dill, and J. Levitt, "Validity Checking for Combination of Theories with Equality," in *Proceedings of FMCAD*, 1996.

[12] 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 *CAV*, 2002.

[13] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with Exhaustive Theory Propogation and its Application to Difference Logic," in *CAV*, 2005.

[14] C. Wang, F. Ivancic, M. Ganai, and A. Gupta, "Deciding Separation Logic Formulae with SAT by Incremental Negative Cycle Elimination," in *Proceeding of Logic for Programming, Artificial Intelligence and Reasoning*, 2005.

[15] M. Ganai, M. Talupur, and A. Gupta, "SDSAT: Tight Integration of Small Domain Encoding and Lazy Approaches in a Separation Logic Solver," 2006.

[16] L. d. Moura, H. RueB, and M. Sorea, "Lazy Theorem Proving for Bounded Model Checking over Infinite Domains," in *Proceedings of CADE*, 2002.

[17] A. Armando, J. Mantovani, and L. Platania, "Bounded Model Checking of Software using SMT Solvers instead of SAT solvers," University of Genova 2005.

[18] M. R. Prasad, A. Biere, and A. Gupta, "A survery of recent advances in SAT-based formal verification.," in *STTT 7(2)*, 2005.

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

[20] M. Ganai and A. Kuehlmann, "On-the-Fly Compression of Logical Circuits," in *Proceedings of International Workshop on Logic Synthesis*, 2000.

[21] J.-C. Filliatre, S. Owre, H. RueB, and N. Shankar, "ICS: Integrated Canonizer and Solver," in *Proceedings of CAV*, 2001.

[22] M. Ganai, A. Gupta, and P. Ashar, "Beyond Safety: Customized SAT-based Model Checking," in *Proceeding of DAC*, 2005.

[23] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of State-based Models," in *Proceedings of ICSM*, 2003.

[24] F. Ivancic, J. Yang, M. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Bounded Model Checking for Software," in *Proceedings of ISOLA*, 2004.

[25] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang, "Using Statically Computed Invariants inside the Predicate Abstraction and Refinement Loop," in *Proceeding of CAV*, 2006.

[26] M. Ganai, A. Mukaiyama, A. Gupta, and K. Wakabayashi, "Another Dimension to High Level Synthesis: Verification," in *Proceedings of Workshop on Designing Correct Circuits*, 2006.