Efficient State Space Exploration: Interleaving Stateless and State-based Model Checking

Malay K. Ganai
NEC Laboratories America

Chao Wang
NEC Laboratories America

Weihong Li
NEC Laboratories America

***Abstract*—State-based model checking methods comprise computing and storing reachable states, while stateless model checking methods directly reason about reachable paths using decision procedures, thereby avoiding computing and storing the reachable states. Typically, state-based methods involve memory-intensive operations, while stateless methods involve time-intensive operations. We propose a *divide-and-conquer* strategy to combine the complementary strengths of these methods for efficient verification of embedded software. Specifically, our model checking engine uses both state decomposition and state prioritization to guide the combination of a Presburger arithmetic based symbolic traversal algorithm (state-based) and an SMT based bounded model checking algorithm (stateless). These two underlying algorithms are interleaved—based on memory/time bounds and dynamic task partitioning—in order to systematically explore the state space and to avoid storing the entire reachable state set. We have implemented our new method in a tightly integrated verification tool called HMC (Hybrid Model Checker). We demonstrate the efficacy of the proposed method on some industry examples.**

I. Introduction

Model checking [1]–[4] plays an important role today in the automated verification of hardware [5] and safety-critical embedded software [6]–[9]. In this paper, we focus on model checking techniques that are based on symbolic reasoning. Based on whether they compute and store the reachable states, symbolic model checking techniques can be classified as either *state-based* or *stateless*.

State-based model checking methods, e.g. those based on Boolean level reasoning using BDDs [2] and integer level reasoning using Presburger arithmetic solvers [11]–[13], comprise computing and storing sets of states. Given the transition relation of the model and a set of states, they use image computations to repeatedly obtain a set of states reachable in either the forward (*image*) or the backward (*pre-image*) directions. This process is often referred to as *unbounded* model checking due to its capability of exploring the entire state space without a predetermined bound on the execution depth. State-based methods generally perform well when the state sets have compact representations, but can suffer from memory explosion otherwise.

Stateless model checking methods, e.g. SAT based bounded model checking (BMC) [3], [4] and high-level bounded model checking [14], [15] based on Satisfiability Modulo Theory (SMT) solvers (e.g. [16], [17]) directly reason about the execution paths of the model, thereby avoiding the explicit manipulation of state sets. Typically memory explosion is less severe; however, stateless methods often involve time-intensive computations. They generally perform well when the execution paths required to decide the validity of the properties are short, but can perform poorly when paths are long.

In this paper, we propose a unified model checking framework for verifying embedded software called HMC (*Hybrid Model Checking*), which combines the often complementary strengths of the state-based [12], [13] and stateless [14], [15] methods. The goal is to provide a robust verification solution for industrial-strength embedded software applications. In our unified framework, as illustrated in Fig. 1 (a), the state traversal (`ST`) and the path-based reasoning (`PR`) are interleaved seamlessly in order to systematically search the state space of the model, at the same time avoiding the storage of the entire (explored) reachable state set. More specifically, at any point of time, the hybrid procedure stores only a set of *frontier states*, as illustrated in Fig. 1 (b), which are the states reachable from the initial states in a fixed number of execution steps.

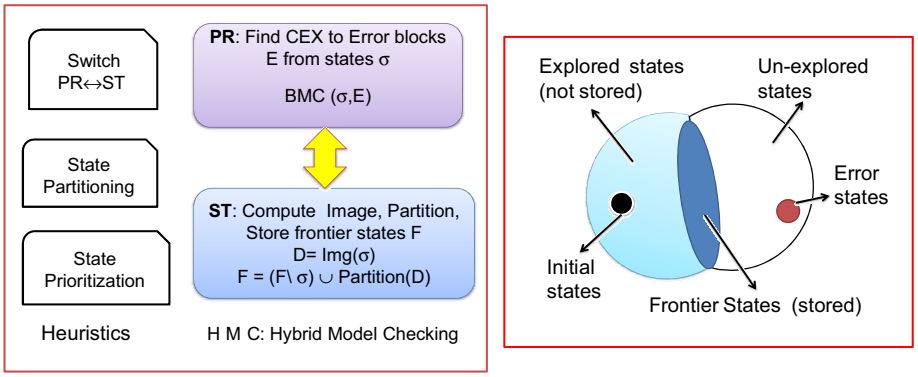
We focus on verifying reachability properties of sequential programs under the assumptions of finite recursion and finite data. Both user-defined assertions and common programming errors (array bounds violations, null pointer de-referencing, uninitialized variables, etc.) can be formulated into the reachability of some *error blocks*. Our HMC procedure is designed specifically to target bugs that are hard-to-find by either state-based or stateless methods in isolation.

The search engine of HMC implements a state traversal algorithm [12], [13] based on a Presburger arithmetic solver [10], and a SMT-based bounded model checking algorithm [14], [15]. The finite-state model is represented as combination of a set of linear integer constraints whenever possible, as opposed to the more conventional, bit-blasted, pure Boolean logic. Our choice of using higher level reasoning rather than pure Boolean logic is because, as shown in [12]–[15], for example, it tends to scale much better when applied to models of embedded software. Since state-based methods typically involve memory-intensive operations, regardless of the logic levels to which they are applied, whereas stateless methods typically involve time-intensive operations, we have designed a heuristic algorithm to automatically interleave the state traversal and the path-based reasoning. To improve the overall robustness, we use a memory bound to switch from `ST` to `PR` and use a depth/time bound to switch from `PR` back to `ST`; we repeatedly switch between these two modes to avoid blowup within a particular method.

We use both *state partitioning* and *state prioritization* to guide the interleaving of state traversal and path-based reasoning in our hybrid search. Large frontier state sets are partitioned dynamically based on heuristics with respect to their size, in terms of the number of linear equations (for Presburger arithmetic) and BDD nodes. The partitioned state sets are ranked to decide their processing order, and the ranking functions are designed to favor those with a higher likelihood of reaching the error states.

The divide-and-conquer style of our HMC procedure introduces data parallelism, which can be exploited by parallelization for many-core CPUs. Below are some highlights of HMC:

- It employs a *divide-and-conquer* procedure to compute the frontier states and to represent them efficiently.
- Although it has avoided storing the entire reachable state set, for terminating programs, it still guarantees exhaustive coverage.
- It employs abstract interpretation to over-approximate the set of states backward-reachable from error states in k steps, allowing the search to target states likely to lead to errors quicker.
- It employs control-state reachability to build simplified transition relations *on-the-fly*, to guide both `ST` and `PR`.



(a) (b)

Fig. 1. (a) Overview of the HMC procedure; and (b) the set of frontier states

.

The remainder of this paper is organized as follows. After establishing notation in Section II and review the basics of symbolic computation in Section III, we present our main contributions in Sections IV-VI. This is followed by the experimental results in Section VII, and the review of related work in Section VIII. We give our conclusions in Section IX.

II. Preliminaries

We briefly discuss the essential model building steps under the typical assumptions of bounded heap/stack for embedded software.

Consider the C program in Fig. 2(a), which has a while loop and some embedded assertions. Fig. 2(b) shows the corresponding control and data flow graph. The boxes associated with control states (i.e. basic blocks) show their unique *id*'s. An edge between blocks corresponds to the control flow between associated program points. Each edge is associated with an enabling predicate. Updates of program variables (i.e. assignments) are shown at each control state. Embedded assertions and common programming errors, such as array bounds violation, null pointer dereferencing, and failed assertion, can be modeled as the reachability of some error blocks. In this figure, for example, blocks 12 and 14 correspond to the assertions $P1$ (line 18) and $P2$ (line 19), respectively.

A. Modeling C Programs as EFSMs

An Extended Finite State Machine (EFSM) is a 3-tuple $M = (\mathcal{C}, \mathcal{X}, \mathcal{T})$, where $\mathcal{C}$ is a set of control states, $\mathcal{X}$ is an n-dimensional space of valuations of the datapath variables, and $\mathcal{T}$ is a set of transitions. Each transition is a 4-tuple (c_i, d_i, c_j, d_j), where $c_i, c_j \in \mathcal{C}$ are control states, and $d_i, d_j \in \mathcal{X}$ are valuations of datapath variables. We use C to denote the program counter variable which takes values from the set $\mathcal{C}$. We use $X = \{x_1, \cdots, x_n\}$ to denote the set of datapath variables, which take values from the set $\mathcal{X}$. A *state* of M is an ordered pair $\langle c_i, d_i \rangle \in \mathcal{C} \times \mathcal{X}$. Let $g_{ij} : \mathcal{X} \mapsto B = \{0, 1\}$ be the guarded transition predicate associated with the transition from c_i to c_j. Let $u_i : \mathcal{X} \times I \mapsto \mathcal{X}$ be the update transition relation associated with the assignments in c_i. Let $\Gamma : \mathcal{C} \times \mathcal{C} \mapsto \{0, 1\}$ be the Boolean predicate such that, for $c_i, c_j \in \mathcal{C}$, $\Gamma(c_i, c_j) = true$ iff g_{ij} is defined.

We construct a symbolic transition relation for the EFSM to capture the set of all guarded transitions between basic blocks. For each data variable, we add an update transition relation based on the expressions assigned to the variable in various basic blocks. Recall that C is the program counter variable and X is the set of datapath variables. We shall use C' and X' to denote the next-state copies of C and X, respectively. Furthermore, let $X = X^B \cup X^I$, where X^B and X^I are the subsets of Boolean variables and integer variables in X, respectively. We use *Boolean* and *Linear integer arithmetic* expressions to represent the update and guarded transition relations of X^B and X^I, respectively.

A transition from $\langle c_i, d_i \rangle$ to $\langle c_j, d_j \rangle$ under enabling predicate g_{ij} and update relation u_i with n assignments is denoted $\langle c_i, d_i \rangle \xrightarrow{g/u} \langle c_j, d_j \rangle$. Let T_{ij} be its transition relation. The transition relation of the entire model M, denoted $T(C, X, C', X')$, is the union of all these individual transitions.

$$T \stackrel{def}{=} \bigvee_{\Gamma(c_i,c_j)=true} \underbrace{(C = c_i \wedge C' = c_j \wedge g_{ij} \wedge u_i)}_{T_{ij}} \tag{1}$$

Let $x_k := e_{i,k}$ be an assignment to variable x_k in control state c_i. Let x'_k be the next-state variable of x_k. The update relation u_i is defined as follows:

$$u_i \stackrel{def}{=} (C = c_i) \wedge \bigvee_{k=1}^{k=n} (x'_k = e_{i,k}) \tag{2}$$

We assume $x'_k = x_k$ if variable x_k is not explicitly updated.

The control flow graph (CFG), denoted $G = (V, E, r)$, can be viewed as the control-state abstraction of the EFSM. The set V of nodes corresponds to the set of control states in the EFSM. E is the set of control flow edges. $r \in V$ is the entry block. The CFG can be obtained from an EFSM by ignoring all enabling predicates and updated transitions. More formally, we assume that $V = C$, $E = \{(c, c') \mid \Gamma(c, c') = true\}$, and r is the unique entry block with no incoming transition.

B. Symbolic Expressions and Solvers

We represent both the transition relations and the state sets symbolically as logic formulas, with Boolean level expressions as well as integer expressions in Presburger arithmetic. Presburger arithmetic is a decidable fragment of quantifier-free first-order logic.

Solving Decision Problems using SMT: A Satisfiability Modulo Theory (SMT) problem for a theory $\mathcal{T}$, denoted $SMT(\mathcal{T})$, comprises a formula with arbitrary Boolean combination of a set of elementary constraints, each of which is expressed in the theory $\mathcal{T}$. For example, if $\mathcal{T}$ is the theory of linear integer arithmetic ($\mathcal{LIA}$), then each elementary constraint is of the form $(a_1 x_1 + \ldots a_n x_n \leq c)$, where $a_1, \ldots, a_n$ and c are integer constants, and $x_1, \ldots, x_n$ are integer

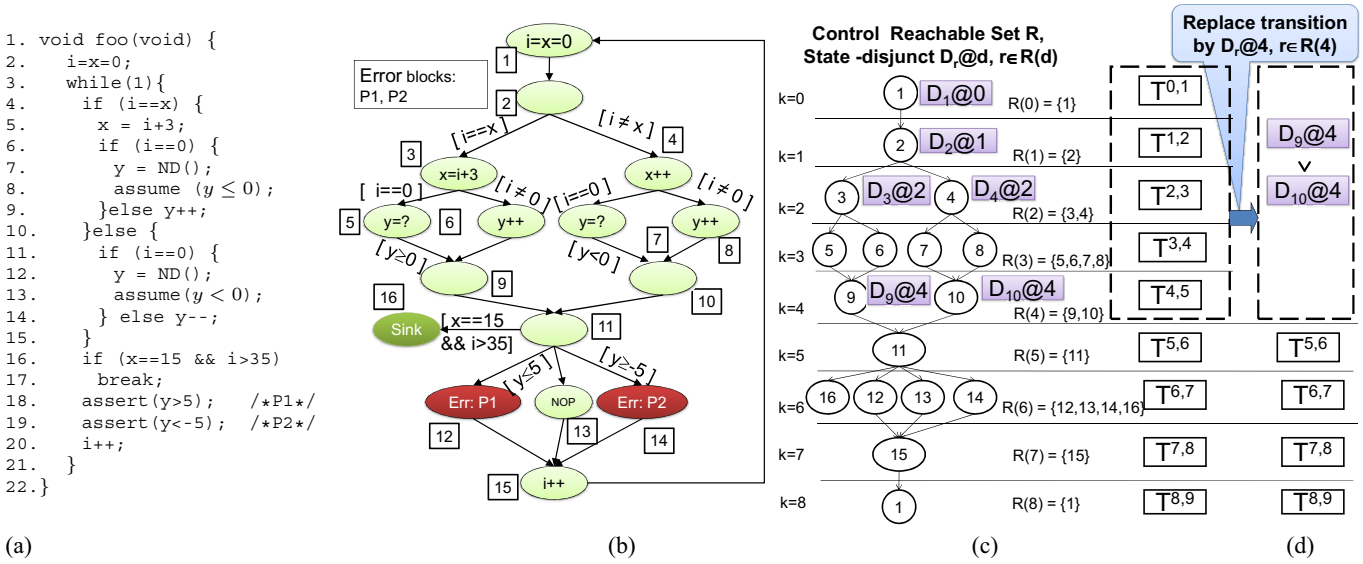


Fig. 2. (a) An example C program, (b) its EFSM M, (c) the CSR for depth 8, and (d) the replacing transition $T^{0,5}$ by a reachable set.

variables. Given an SMT formula ϕ, the decision problem is to determine whether ϕ is $\mathcal{T}$-satisfiable, i.e., there exists a valuation of the Boolean and integer variables in ϕ such that ϕ evaluates to true. In this work, we use an off-the-shelf SMT solver [16] to decide the formulas in SMT modulo linear integer arithmetic ($\mathcal{LIA}$).

Simplifying Presburger formulas: We use a Disjunctive Normal Form (DNF) representation for the formulas representing state sets, i.e., formula Ω expressed as $\Omega = \bigvee_{i=1}^{n_\Omega} \Omega_i^B \wedge \Omega_i^I$, where Ω_i^B is a Boolean formula, Ω_i^I is an integer formula in Presburger arithmetic, and n_Ω is the number of disjunctions, respectively. Let v^I be the set of integer variables and v^B be the sets of Boolean variables. We assume that the formulas, i.e. Ω_i^I and Ω_i^B, are always type-consistent; that is, $v^B \cap v^I = \emptyset$.

Consider two DNF formulas Ω and $f = \bigvee_{i=1}^{n_f}(f_i^B \wedge f_i^I)$. Common set operations such as union ($\cup$), conjunction ($\cap$), negation ($\neg$), and existential quantification ($\exists v.$) can be defined over DNF formulas [11], [12]. The union of two DNF formulas is simply the union of their subformulas. The conjunction is the union of the pair-wise conjunctions of their subformulas.

$$\Omega \wedge f = \bigvee_{i=1,j=1}^{n_\Omega,n_f} (\Omega_j^B \wedge f_i^B) \wedge (\Omega_j^I \wedge f_i^I)$$

Since there is no common variable, subformulas from different domains do not interfere with each other. The negation of a DNF formula is implemented in a similar way. Since there is no common variable, existential quantification distributes not only over unions (as in general) but also over subformulas in different domains:

$$\exists v^B, v^I \ . \ \Omega = \bigvee_{i=1}^{n_\Omega} (\exists v^B \ . \ \Omega_i^B) \wedge (\exists v^I \ . \ \Omega_i^I)$$

The DNF representation is not a canonical form, and there exist heuristic algorithms [11], [12] to compact the result. Although the resulting DNF can be $(n_\Omega \times n_f)$ for conjunction (and 2^{n_Ω} for negation), such worst-case results rarely happen in practice. In our implementation, we use CUDD [18] to represent and simplify Boolean formulas, and the Omega library [10] to represent and simplify Presburger arithmetic formulas.

III. Symbolic Computations

A. Path-based Reasoning

Let $s_i \equiv \langle C, X \rangle$ denote a symbolic state. A path is a finite sequence $\pi^{0,k} = (s_0, \ldots, s_k)$ satisfying the following predicate:

$$T^{0,k} \stackrel{def}{=} \bigwedge_{0 \le i < k} T(s_i, s_{i+1}) \tag{3}$$

where T is the transition relation as in Eq. (1). Let $T^{0,0} \stackrel{def}{=} true$. In bounded model checking, whether an LTL property ϕ can be falsified in k execution steps from some initial state ψ is formulated as a satisfiability problem [3]:

$$BMC^k(\psi, \phi) \stackrel{def}{=} \psi(s_0) \wedge T^{0,k} \wedge \neg\phi(s_k) \tag{4}$$

where $\phi(s_k)$ means that ϕ holds in state s_k, and $\psi(s_0)$ means ψ holds in state s_0. Given a predetermined bound n, BMC iteratively checks the satisfiability of BMC^k for $0 \le k \le n$ using a SAT or SMT solver. We define $BMC^{k,k+d}$ between depths k and $k+d$ as

$$BMC^{k,k+d}(\psi, \phi) \stackrel{def}{=} \psi(s_k) \wedge T^{k,k+d} \wedge \neg\phi(s_{k+d}) \tag{5}$$

For each variable $v \in X$, let v^k be the copy of v at the BMC unrolling depth k. When considering the reachability of error block Err from a source block Src, we define $\psi := (C = Src)$ and $\neg\phi := \mathbf{F}(C = Err)$, where $\mathbf{F}$ is the LTL operator *eventually*.

B. State Traversal

Given the entry block $c_0 \in \mathcal{C}$ and initial values for datapath variables $(x_1 = e_{0,1}; \cdots x_n = e_{0,n})$, the initial state formula I is

$$I \stackrel{def}{=} (C = c_0) \wedge \bigwedge_{k=1}^{n} (x_k = e_{0,k}) \tag{6}$$

Similarly, given a subset $\rho \subseteq \mathcal{C}$ of control states, the set $D_\rho(C, X)$ of concrete states in ρ is defined as follows:

$$D_\rho \stackrel{def}{=} \bigvee_{c \in \rho} (C = c) \wedge f_c^B \wedge f_c^I \tag{7}$$

where f_c^B and f_c^I denote the subformulas over datapath variables in Boolean logic and Presburger arithmetic, respectively. We refer

to a state set like D_ρ as a *state-disjunct*. Given a transition relation $T(C, X, C', X')$ and a state-disjunct $D_\rho(C, X)$, symbolic state traversal typically comprises a series of image computations [12] till a fixpoint is reached. Let $f_{(X/X')}$ denote the substitution of X' variables in function f by the corresponding X. The image operation Img, which computes the set of states reachable from D_ρ in one execution step, is defined as:

$$Img(D_\rho) \stackrel{def}{=} (\exists_{C,X}\ T \wedge D_\rho)_{(X/X',C/C')} \tag{8}$$

Similarly, the pre-image operation Pre is defined as:

$$Pre(D_\rho) \stackrel{def}{=} (\exists_{C',X'}\ T \wedge (D_\rho)_{(X'/X,C'/C)}) \tag{9}$$

We use $Img^k(D_\rho)$ and $Pre^k(D_\rho)$ to denote the result of k successive Img and Pre operations from D_ρ, respectively. We use $Pre^+(D_\rho)$ to denote the result of an over-approximated pre-image operator [19], which is a superset of states backward-reachable from D_ρ in one step.

C. Control State Reachability

A control path $\gamma^{0,k} = (c_0, \ldots, c_k)$ is a sequence of control states satisfying the following predicate:

$$\Gamma^{0,k} \stackrel{def}{=} \bigwedge_{0 \le i < k} \Gamma(c_i, c_{i+1}) \tag{10}$$

Recall that $\Gamma(c_i, c_j) = true$ iff the guarded transition g_{ij} is defined for c_i and c_j. Let $\Gamma^{0,0} = true$. A control state reachability (CSR) analysis is a breadth-first traversal of the CFG where control state b is one step reachable from a iff $\Gamma(a, b) = true$, i.e,

$$CSR(\tilde{c}) \stackrel{def}{=} \{b | a \in \tilde{c}, \Gamma(a, b) = true\} \tag{11}$$

We use $CSR^d(\tilde{c})$ to denote the result of d successive CSR operations from a set $\tilde{c}$ of control states. At an execution depth d, let $R(d)$ be the set of control states reachable from $R(d-1)$ in one step in the CFG by ignoring the guards. Let $R(0) = c_0$, then $R(d) = CSR^d(R(0))$. Consider computing CSR for the CFG in Fig. 2 (b). The resulting set $R(d)$ is shown in Fig. 2 (c). We will use control state reachability to simplify the formulas in both state-based and stateless computations.

IV. HMC: The Overview

We illustrate the basic ideas behind the hybrid model checking in this section, and provide formal exposition in Sections V- VI.

A. Basic Strategy

Consider the example in Fig. 2 (a-b). We first obtain an unrolled CFG by unwinding the CFG up to depth $k = 8$ as shown in Fig. 2 (c). Each program (concrete) path in the depth-k BMC instance corresponds to a control (abstract) path in this unrolled CFG. As the unrolling depth increases, the number of control paths will increase quickly, thereby making each successive BMC instance harder to solve. As shown in Fig. 2 (c), the number of control paths from block 1 at $k = 0$ to block 12 at $k = 6$ is 4, but is only 1 from block 11 at $k = 5$. Clearly, if we can replace the unrolled transitions $T^{0,4}$ in the dotted rectangle by the set of concrete reachable states at $k = 4$, we can reduce the size of the BMC instances for $k > 4$, thus making BMC search faster and deeper. Let $D_c@k$ be a state-disjunct in control state c at depth k from the initial states. We replace $T^{0,4}$ by $D_9@4 \vee D_{10}@4$.

***Definition** 1 (Frontier Set):* Let F be a set of state-disjuncts reachable from the initial states. F is a frontier set with respect to the error states B_{Err} (in the error block Err) iff some error states in B_{Err} are reachable from F in k steps; that is, there exists $D \in F$ such that $B_{Err} \cap Img^k(D) \neq \emptyset$, with $Img^0(D) \stackrel{def}{=} D$.

***Lemma** 1:* If F is a frontier set with respect to B_{Err} and $B_{Err} \cap D = \emptyset$, then $F := F \backslash \{D\} \cup \{Img(D)\}$ is also a frontier set.

Essentially, F maintains a set of state-disjuncts that divides the reachable state space into explored and unexplored as in Fig. 1 (b).

In HMC, we pick the next to-be-explored state-disjunct $D_\rho \in F$ using a state prioritization heuristic (Section V-B). The set D_ρ is then encoded as a SMT($\mathcal{LIA}$) formula sc, to be used as reachable state constraints in $BMC^{k,k+d}(\psi, \phi)$. Based on the disjunctive partition of F, we can solve a series of independent satisfiability problems given by

$$\omega \stackrel{def}{=} BMC^{k,k+d}(\psi, \phi) \wedge (\psi = sc) \tag{12}$$

where each sc is the constraint for a small subset of the frontier F computed symbolically. More specifically, we employ the following strategies in HMC to avoid the memory blowup:

- We store only the frontier sets, not the entire set of reached states.
- When the size of a state-disjunct $\in F$ exceeds a threshold, we partition it further (Section V-A) to make its image computation less memory-intensive.
- For computing $Img(D_\rho)$, we build a simplified transition relation $T|_{D_\rho}$[1] on the fly and release its memory immediately after the computation (ref. Section V).

We switch between computing reachable states and solving the satisfiability problem based on some predetermined memory/time bounds.

B. Completeness Discussion

In HMC we do not accumulate the entire set of reachable states, therefore avoiding the main memory bottleneck in symbolic state traversal. This also means, in general, that we cannot guarantee to detect the reachable fixpoint. Fortunately, if the programs under verification are terminating programs, state traversal with frontier sets is guaranteed to terminate after L steps, where L is the longest program path [12].

For path-based reasoning, we can also obtain a completeness criteria [3], [4] from the BMC completeness threshold $\mathcal{CT}$. For example, for a state set ψ, we know that the threshold $\mathcal{CT}$ is reached if the formula below becomes unsatisfiable:

$$LFP^{k,k+d} \stackrel{def}{=} BMC^{k,k+d}(\psi, \phi) \wedge \bigwedge_{k \le i < j \le k+d} (s_i \neq s_j) \tag{13}$$

In our HMC implementation, we also use the more recent improvement in [15], which requires solving a formula of only size $O(d)$ as opposed to the conventional $O(d^2)$. The key is to note that for a terminating problem, the SINK control state (exit block) does not have outgoing transitions.

$$NTP^{k,k+d} \stackrel{def}{=} (BMC^{k,k+d}(\psi, \phi) \wedge (\phi = B_{SINK})) \tag{14}$$

***Lemma** 2:* For terminating programs, $NTP^{k,k+d}$ is satisfiable if and only if $LFP^{k,k+d+1}$ is satisfiable.

[1]For f and g, we define a constraint (or simplify) operator, denoted $f|_g$, so that $f|_g = f$ if $g = 1$; otherwise, $f|_g = don't\_care$. Thus, $f|_g \wedge g = f \wedge g$.

V. State Decomposition and Prioritization

We use the cheap control state reachability analysis to simplify the subsequent precise image computations in state traversal. Given the transition relation T and a state-disjunct $D_\rho(C, X)$ with the corresponding control states $\rho \subseteq \mathcal{C}$, conceptually we can obtain $T|_{D_\rho}$ by projecting T to D_ρ,

$$T \wedge D_\rho = T|_{D_\rho} \wedge D_\rho = \underbrace{(\vee_{c_i \in \rho, c_j \in CSR(\rho)} T_{ij})}_{T_{\rho, CSR(\rho)}} \wedge D_\rho \quad (15)$$

The number of disjuncts in $T_{\rho,CSR(\rho)}$ is bounded by the number of distinct control edges. Typically the number of outgoing edges from a basic block is either 1 or 2 (for if-else); therefore the number of disjuncts is between $|\rho|$ and $2|\rho|$. In contrast, the number of disjuncts in T, as in equation (1), is bounded by total number of control edges in the CFG. Thus, instead of building T, we build $T_{\rho,CSR(\rho)}$ on the fly for each given D_ρ using the CSR information. Since the symbolic representation of $T_{\rho,CSR(\rho)}$ is often smaller than that of T, the peak memory requirement during the state traversal can be reduced significantly. Further, since CSR is an over-approximated analysis, the precise image $Img(D_\rho)$, where D_ρ is a subset of the set ρ of control states, is always a subset of $CSR(\rho)$ as stated in the following:

Lemma 3: Let $D'_{\rho'} = Img(D_\rho)$. If $\rho \in R(k)$, then $\rho' \subseteq CSR(\rho) \subseteq R(k+1)$.

Lemma 4: If $D_\rho \subseteq Img^k(I)$, then $\rho \subseteq R(k)$.

A. State Decomposition

Our DNF representations in general are not canonical. In particular, the polyhedrons used to represent linear integer constraints may become fragmented after being propagated through branching and re-converging points of the CFG during image computation. This often lead to memory blow up when verifying large programs. We mitigate this problem by dynamically decomposing the frontier sets.

The goal is to avoid computing images/pre-images for large state representations. Given a state-disjunct D_ρ and its disjunction $\{D_{1,\rho_1}, \cdots D_{i,\rho_m}\}$ s.t. $D_\rho = \bigvee_i^m (D_{i,\rho_i}), \rho = \bigcup_i^m \rho_i$, we divide the image computation into multiple steps to avoid memory blowups,

Lemma 5: $Img(D_\rho) = \bigvee_{i=1}^m Img(D_{i,\rho_i})$

In general computing a good state partition is hard due to often conflicting requirements. Larger partitions can make each subproblem more difficult to solve but can reduce the partitioning overhead. Whereas smaller partitions can result in easier image computation subproblems but can significantly increase the partitioning overhead. Here we use low-overhead partitioning heuristics based on the CSR information. We note that the size of $Img(D_\rho)$ often depends on the sizes of both T and D_ρ. We want to partition D_ρ but want to minimize the need for partitioning $Img(D_\rho)$. We propose a simple mechanism to estimate the size of a state-disjunct and its image.

Given $D_\rho = \bigvee_{c_i \in \rho} (C = c_i) \wedge f_i^B \bigwedge f_i^I$, we define $size(D_\rho)$ as

$$size(D_\rho) \stackrel{def}{=} \Sigma_{c_i \in \rho} (\#P(f_i^I) + \#B(f_i^B)) \quad (16)$$

where $\#P$ is the number of polyhedra in f_i^I, and $\#B$ is the number of BDD nodes in f_i^B. We estimate the size of $Img(D_\rho)$ before actually computing it. The size, denoted $size(TD_\rho)$, is defined based on the size of $T_{\rho,CSR(\rho)} \wedge D_\rho$:

$$\begin{aligned} size(TD_\rho) \stackrel{def}{=} & \Sigma_{c_i \in \rho} \#P(f_i^I \wedge u_i^I) * (\Sigma_{c_j \in CSR(\rho)} \#P(g_{ij}^I)) \\ & + \Sigma_{c_i \in \rho} \#B(f_i^B \wedge u_i^B) * (\Sigma_{c_j \in CSR(\rho)} \#B(g_{ij}^B)) \end{aligned} \quad (17)$$

Given D_ρ, we define a *no-need-to-partition* predicate $NP(D_\rho)$:

$$NP(D_\rho) \stackrel{def}{=} (size(D_\rho) < t_{size}) \wedge (size(TD_\rho)) < i_{size}) \quad (18)$$

The partitioning problem is stated as follows. Given the threshold sizes t_{size} and i_{size} and a state-disjunct D_ρ, divide D_ρ into a set $\{D_{1,\rho_1}, \cdots D_{m,\rho_m}\}$ such that, for $1 \leq i \neq j \leq m$, $\rho_i \subseteq \rho$, $NP(D_{i,\rho_i})$ is $true$, and m is minimum. We use two low-overhead greedy procedures, *Partition* and *Group*, which are geared toward minimizing the number of partitions and at the same time obtaining a good balance among the partition sizes.

Partition: Given D_ρ s.t $NP(D_\rho) = false$.

- Initialize $H = \{D_{1,\rho_1}, \cdots, D_{n,\rho_n}\}$, s.t. $\rho_i = \{c_i\}$, $c_i \in \rho$.
- For each $D_{i,\rho_i} \in H$, if $NP(D_{i,\rho_i}) = false$, bi-partition D_{i,ρ_i} greedily s.t. $D_{i,\rho_i} = D_{a,\rho_i} \wedge D_{b,\rho_i}$. Update H, i.e., $H := H \backslash \{D_{i,\rho_i}\} \cup \{D_{a,\rho_i}, D_{b,\rho_i}\}$.
- Repeat till all partitions in H satisfies predicate NP.

Group: Given a set $J = \{D_{1,\rho_1}, \cdots, D_{m,\rho_m}\}$ s.t. $\rho_i \in R(k)$.

- Merge two smallest state-disjuncts in J, i.e. $D_{12,\rho_1 \cup \rho_2} = D_{1,\rho_1} \vee D_{2,\rho_2}$. (Implemented using priority-queues.)
- Update $J = J \backslash \{D_{1,\rho}, D_{2,\rho}\} \cup \{D_{12,\rho_1}\}$ and repeat the previous step if $NP(D_{12,\rho_1 \cup \rho_2}) = true$; otherwise stop.

B. State Prioritization

We propose two heuristics, one is goal-directed and another is driven by the search depth, to prioritize the processing of state-disjuncts in the frontier set F. Following the terminology of [21], we use *lighthouses* to refer the intermediate states serving as guidance of the search for error states.

Goal-directed lighthouses (`glh`). These lighthouses are derived based on the observation that a state is more likely to reach an error block if the block is only few execution steps away. We compute the over-approximated execution steps using the Pre^+ operator (below). Let BR^k be a set of abstract (backward) states obtained by applying Pre^+ operator k times from the error block. Then each state-disjunct $D \in F$ is given an *abstract distance* k iff $D \cap BR^k \neq \emptyset$ and $\forall_{0 \leq i < k}$, $D \cap BR^i = \emptyset$. Our `glh` strategy is to give $D_1 \in F$ a higher priority over $D_2 \in F$ if the abstract distance of D_1 is less than that of D_2.

Pre^+ operator: From a given state disjunct D, we first compute its pre-image using equation (9), then obtain an over-approximation by computing the convex union of the disjuncts, i.e., $(Pre(D))^+$. Let K be a predetermined threshold for the maximal number of polyhedra allowed in each state-disjunct. If the actual number of polyhedra in $Pre(D)$ exceeds K, we replace the *Polyhedra-part* of the state-disjunct with at most K number of polyhedra, the later of which is computed by heuristically merging the polyhedra as in [12], [13].

Example: Let $K = 1$, and the state disjunct D be

$$\begin{aligned} D = \quad & (C = 0) \wedge (x = 0) \wedge (y = 0) \vee (C = 1) \wedge \\ & (\ (x \geq 1) \wedge (x \leq 5) \wedge (y \geq 2) \vee (x \geq 10) \wedge (y \geq 1)\ ) \end{aligned}$$

where the expressions $C = 0$ and $C = 1$ are in Boolean logic and are represented as BDDs. The over-approximated set D^+ is

$$(C = 0) \wedge (x = 0) \wedge (y = 0) \vee (C = 1) \wedge (x \geq 1) \wedge (y \geq 1)$$

Targeting deeper states (`dfs`): To target deep error blocks, we give priority to the state-disjuncts at a deeper execution depth. Let $D_{\tilde{c}_k} \in F$ be the set reachable from the initial states in the k execution steps. In the `dfs` strategy, we give a higher priority to state-disjunct $D_{\tilde{c}_k} \in F$ over priority over $D_{\tilde{c}_j} \in F$, iff $k > j$.

C. BMC Simplification

We also use CSR information to simplify the BMC instances. The set of control paths of length d from $c \in \tilde{c}_k (\subseteq R(k))$ is

$$\tilde{\gamma}_{\tilde{c}_k,d} = \{(c_k \cdots c_i \cdots c_{k+d}) \mid \forall_{k \le i \le k+d}\ c_i \in CSR^i(\tilde{c}_k)\} \quad (19)$$

We constrain $BMC^{k,k+d}$ using $\tilde{\gamma}_{\tilde{c}_k,d}$; that is, $BMC^{k,k+d}|_{\tilde{\gamma}_{\tilde{c}_k,d}}$, to remove the statically unreachable control paths:

$$BMC^{k,k+d}|_{\tilde{\gamma}_{\tilde{c}_k,d}}(\psi,\phi) \stackrel{def}{=} T^{k,k+d}|_{UBC(\tilde{\gamma}_{\tilde{c}_k,d})} \wedge \psi \wedge \neg\phi(s_{k+d}) \quad (20)$$

where UBC, or Unreachable Block Constraint, uses unreachable control states to simplify $T^{k,k+d}$ using on-the-fly size reduction techniques such as hashing and constant folding [14]. Note that B_r is a control-state predicate, i.e., $B_r \equiv (C = r)$.

$$UBC(\tilde{\gamma}_{\tilde{c}_k,d}) \stackrel{def}{=} \bigwedge_{0 \le i < d, r \notin CSR^i(\tilde{c}_k)} \neg B_r^{k+i} \quad (21)$$

***Lemma** 6:* $BMC^{k,k+d}|_{\tilde{\gamma}_{R(k),d}} \Longleftrightarrow_{SAT} BMC^{k,k+d}$.

VI. HMC: The Procedure

The pseudo code of our HMC procedure is shown in Algorithm 1. Given an error block Err and the initial states I, the procedure interleaves subprocedures PR and ST till Err is proved to be either reachable or unreachable from I. The frontier set F of state-disjuncts is stored in a priority queue (Section V-B). F is initialized to I.

Subprocedure **PR**. It always chooses the state $D_{\tilde{c}_k} \in F$ with the highest priority. For $k \le i \le k+d$, it first computes $CSR^i(\tilde{c}_k)$ for a fixed depth d. It then obtains the $BMC^{k,i}$ instance as follows: (a) for each i, build a simplified transition relation $T^{k,k+i}$ (Section V-C), and (b) translate $D_{\tilde{c}_k}$ into a state constraint sc at depth k. For each $BMC^{k,i}$ instance, it performs following two checks. When the depth/time bound is reached, we switch from PR to ST.

Completeness Check: If $SINK \in CSR^i(\tilde{c}_k)$ and if only SINK state (i.e. no non-SINK state) is reachable from $D_{\tilde{c}_k}$, then we can remove state-disjunct $D_{\tilde{c}_k}$ from F as it will never lead to any new state.

Error Check: If $Err \in CSR^i(\tilde{c}_k)$ and if Err is reachable from $D_{\tilde{c}_k}$, then HMC terminates with a return value $FAIL$ as the Err block is reached.

Subprocedure **ST**. If $F = \emptyset$, HMC terminates with a return value $PASS$ since the fixpoint is reached. Otherwise, a switching condition from ST to PR is checked as follows: if the sum of the sizes of all state-disjuncts in F is above a predetermined memory bound f_{size}, then f_{size} is scaled up by factor 2, and simultaneously a switch is made to BMC. Such a switching heuristic allows us to have a finer control over the overall memory usage.

If no switching is needed, it computes a new frontier set as follows: (1) Pick $D_{\tilde{c}_k} \in F$ with the highest priority. If $Err \in \tilde{c}_k$, HMC terminates with $FAIL$; otherwise, obtain a simplified transition relation $T_{\rho,CSR(\rho)}$ (Section V). (2) Compute $img := Img(D_{\tilde{c}_k})$, and if required, partition img further (Section V-A). (3) Remove $D_{\tilde{c}_k}$ from F and then add img or its partitions. If required, also merge small state-disjuncts in F together (Section V-A). The entire process is repeated.

A. Correctness of HMC

***Lemma** 7:* States in F are reachable from the initial state I, i.e., If $D_\rho \in F$, then $D_\rho \in Img^k(I)$ and $\rho \subseteq R(k)$ for some $k \ge 0$.

***Lemma** 8:* F is indeed a frontier set (definition 1), i.e., if Err is reachable from I, then it is reachable from some states in F.

Algorithm 1 HMC

```
1: input: CFG: G, Initial States: I, Transition: T, Error Block: Err
2: output: FAIL/PASS
3:
4: F = I
5: //Switch between PR and ST
6: while true do
7:    PR(G, F, T, Err)
8:    ST(G, F, T, Err)
9: end while

1: Procedure PR (G, F, T, Err) //Perform CSR, BMC
2: //Stop if time bound is reached
3: //Pick a state-disjunct with highest priority
4: for all D_{c̃_k} ∈ F, c̃_k ∈ R(k) do
5:    Compute_CSR(CFG, c̃_k, d) //depth bound is d
6:    for k ≤ i ≤ k + d do
7:       BMC^{k,k+i}_{γ̃_{c̃_k},i} := BMC^{k,k+i}|_{γ̃_{c̃_k},i} //Section V-C
8:       sc = Enc2LIA(D_{c̃_k}) //Encode into SMT
9:       //Completeness: is block other than SINK reachable?
10:      if SINK ∈ R(k) then
11:         is_sat = SAT(BMC^{k,k+i}_{γ̃_{c̃_k},i}(sc, B_SINK))
12:         if (is_sat = false) then
13:            F := F\D_{c̃_k} //Remove dead-end states
14:         end if
15:      end if
16:      if Err ∈ R(k) then
17:         is_sat = SAT(BMC^{k,k+i}_{γ̃_{c̃_k},i}(sc, ¬B_Err))
18:         if (is_sat = true) then
19:            return FAIL //Err is reachable
20:         end if
21:      end if
22:   end for
23: end for

1: Procedure ST (G, F, T, Err) //Compute new frontier set
2:
3: while true do
4:    if F = ∅ then
5:       return PASS //Err is unreachable
6:    end if
7:    Prioritize(F)
8:    if Σ_{d∈F} size(d) > f_size then
9:       f_size := 2 * f_size
10:      break //Switch to PR
11:   end if
12:   if Err ∈ ρ then
13:      return FAIL //Err is reachable
14:   end if
15:   //Pick a state-disjunct with highest priority
16:   T_{ρ,CSR(ρ)} = T|_{D_ρ}
17:   img = Img(D_ρ)
18:   //Move the frontier forward
19:   F := F\D_ρ
20:   F := F∪ Partition(img) //Section V-A
21:   F := Group(F) //Section V-A
22: end while
```

***Lemma** 9:* $\bigvee_{D_\rho \in F} BMC^{k,k+d}(\psi,\phi) \wedge (\psi = D_\rho) \Longleftrightarrow_{SAT} BMC^k(I,\phi)$

***Theorem** 1:* The procedure HMC will terminate eventually with a decision $FAIL$ if the error block is reachable. Further, for a terminating programs, HMC will terminate eventually with a decision $PASS$ if the error block is unreachable.

Proof of termination: From Lemma 8, we have the set F as a frontier set. After every image computation, the frontier set moves closer to the Err block. If the Err block is reachable, the procedure will eventually find it reachable from some state-disjunct in F, and it will terminate. If the Err block is unreachable, the frontier set F will eventually become $\emptyset$ for a terminating program model, and therefore, the procedure will also terminate.

Proof of correctness: Using Lemma 6 we show that PR correctly decides $FAIL$ when the Err block is reachable. Using Lemmas 2 and 6 we show that PR correctly decides unreachability of Err from a given state in F. Using Lemmas 7 and 8 we show that ST correctly decides $FAIL$. Further, when there are no new frontier states, i.e, $F = \emptyset$, it correctly decides $PASS$ as all states have been explored, and no violation was previously reported as per Lemma 9. □

VII. Experiments

We have implemented the HMC algorithm using the SMT solver *yices-1.0.20* [22], the CUDD Library [18], and the Omega library [10]. We obtained EFSMs from C programs of several industry applications as benchmarks, including information management system utilities, ftp utilities, network applications, embedded applications in portable devices, lightweight directory access protocol (LDAP), and Dhrystone benchmark programs. After aggressive program slicing and constant value propagation, each EFSM has control states ranging from 300 to 400. We checked for reachability errors related to array bound violations, pointer validity, memory leaks, and illegal string operations.

We present our experimental results for 23 benchmarks (s1-s23), whose error blocks are known to be hard reach by either state-based methods or path-based methods in isolation. We compare the performance of HMC with a SMT-based BMC algorithm (BMC) and a Presburger arithmetic solver based state traversal algorithm (MIX). For a fair comparison, we have used the same set of EFSM models, the same SMT solvers, the same BDD packages, and the same Presburger solvers.

Our experiments were conducted on a Linux workstation with a 3.4GHz CPU and 2GB of RAM. The time limit is 2000 seconds and the memory limit is 2GB for each run. In HMC, we used 100s time-bound and 30 depth-bound (whichever is reached earlier) to switch from PR to ST. We used $t_{size} = 10$, $i_{size} = 10$, and initial value of $f_{size} = 30$ to switch from ST to PR. We experimented with both the deep-state (dfs) and the goal-directed (glh) prioritization strategies. For glh, we set a time limit of 60s, and a threshold $K = 1$ for the size of polyhedra to compute the abstract states.

The results are shown in Table I. Column 1 shows the benchmark names. Columns 2—4 show the results of BMC, i.e., time (T, in sec), mem (M, in Mb), and witness depth (wd) if a witness is found (highlighted in **bold**); otherwise, the peak depth reached (d*). Columns 5—7 shows the results of MIX. Columns 8—10, 11—13, and 14—16 shows the results of HMC without state partitioning, and the full-blown HMC with two different prioritization strategies dfs and glh, respectively. For BMC and MIX, d* is the maximum search depth for *all paths* from the initial state; for HMC, it is the depth for *some paths* from the initial state.

Overall, HMC with dfs and glh find 19 and 15 witnesses, respectively, while BMC and MIX in isolation can only find 6 and 14 witnesses, respectively. Furthermore, HMC with dfs finds 4 unique witnesses (not found by other methods). We also observe that the MIX times out in several examples, which is due to the time-consuming polyhedral simplification inside the Omega library which gets invoked if the memory usage increases significantly during image computations. The memory usage for BMC is often small. (Witnesses in HMC may be longer since it is not a breadth-first traversal.)

In Table II, we provide statistics for HMC with both dfs and glh. Columns 2–5, and 6–9 present the peak number of switches i.e., PR→ST→PR (#S), the number of image computations (#I), the size of frontier set (#F), and the number of calls to Partition (#P), respectively. For examples such as s10, s11, s12, where glh is able to perform better than dfs, the number of image computations and partitioning is also fewer for glh than that for dfs.

VIII. Related Work

Prior work on integrating multiple verification algorithms can be classified as either *Combination* strategies or *Partition* strategies.

Combination Strategies. In software verification, various complementary techniques, including testing, have been combined to address the scalability problem [21], [23], [24]. Typically, testing is used to reach a concrete state, which is then used as a seed for the subsequent symbolic traversal. In particular, lighthouses have been used to guide the seed selection [21]. Other methods have used target enlargement [25] and retrograde analysis [26]. The goal is to increase the set of states that are known to lead to errors. Over-approximated pre-images of the error states [19] have been used as the enlarged target either to guide simulation [27] or to improve BMC search [28].

In an interpolant-based approach [29], a bounded over-approximated reachable state set is derived using interpolants, and used as initial state constraints for the subsequent SAT-based BMC. The difference is that such initial state constraints are not precise (as is in our case), and therefore needs multiple refinement steps. In [30], a state-based approach using BDDs is applied to obtain a set of reachable set for some bound. This reachable set is then used as an initial state constraint for witness search in a stateless approach using SAT-based BMC. The approach is severely limited by the fact that it invokes BDD-based model checking only once, and therefore may run into severe memory blowup. In contrast, we interleave state traversal with path-based reasoning at finer granularity, rather than invoke the state-based approach only once as in [30].

Partition Strategies. There are prior work on state partitioning, applied both to stateless and state-based model checking algorithms. In CFG-based traversal methods [31], path conditions are generated for a chosen program path in order to execute it symbolically. In a recently improvement [32], BMC instances are partitioned based on the control paths, to disjunctively decompose the problem into smaller subproblems in *tunnels*. The work in [33] decompose BDD representations of reachable states in order to parallelize the fixpoint computation. In several other BDD-based approaches [34]–[36], the transition relations, rather than the state sets, are decomposed disjunctively in order to break down the image computation over smaller components. However, these methods do not repeatedly interleave the state traversal with a path-based reasoning procedure such as PR.

IX. Conclusion

We presented a hybrid model checking algorithm that interleaves a state-based procedure and a stateless symbolic procedure specifically targeted to analyze embedded software. We combine the complementary strengths of both procedures in a tightly integrated framework. We use control flow reachability information to simplify the transition relation for both path-based reasoning and state traversal. We present a simple and yet efficient mechanism to partition state sets for efficient image computation. Furthermore, we present state prioritization techniques that are targeted to find errors quicker. Our experimental

TABLE I
COMPARISON OF HMC WITH BMC(STATELESS), MIX(STATE-BASED)

Ex	BMC [14]			MIX [12]			HMC-no-part			HMC(glh)			HMC(dfs)		
	T	M	d*/wd	T	M	d*/wd	T	M	d*/wd	T	M	d*/wd	T	M	d*/wd
s1	0.1	156	**16**	43	631	**16**	60	560	**16**	95	561	**16**	36	543	**16**
s2	102	156	**55**	TO	1G	31*	606	MO	39*	1.8K	MO	108*	201	738	**59**
s3	1.8K	156	**81**	TO	1G	31*	601	MO	39*	991	1.1G	**87**	562	1G	**90**
s4	1.7K	156	**80**	TO	1G	31*	600	MO	39*	1K	1.1G	**86**	543	1G	**89**
s5	1.1K	156	**75**	TO	1G	31*	592	MO	39*	1K	1.3G	**80**	1.1K	1.6G	**86**
s6	752	156	**69**	TO	1G	31*	589	MO	39*	1K	1.3G	**75**	977	1.3G	**78**
s7	TO	160	71*	TO	1G	31*	593	MO	39*	1.3K	MO	108*	951	1.2G	**76**
s8	TO	160	75*	TO	1G	31*	605	MO	39*	1.7K	MO	51*	1.1K	1.6G	**101**
s9	TO	160	68*	TO	1G	31*	594	MO	39*	1.8K	MO	108*	289	837	**69**
s10	TO	160	114*	69	504	**114**	197	1G	**114**	324	952	**117**	1.3K	1.3G	**220**
s11	TO	160	117*	132	550	**117**	190	1.1G	**117**	334	960	**120**	1.7K	1.3G	**223**
s12	TO	160	120*	189	686	**128**	246	1.2G	**128**	353	1G	**132**	TO	1.6G	258*
s13	TO	160	113*	40	504	**113**	192	989	**113**	TO	MO	521*	TO	1.6G	152*
s14	TO	160	120*	896	1.7G	**180**	525	2G	**180**	TO	MO	550*	TO	1.6G	259*
s15	TO	160	122*	231	711	**131**	260	1.2G	**131**	353	1G	**135**	1.9K	1.6G	**239**
s16	TO	160	114*	69	504	**114**	194	1G	**114**	TO	MO	519*	TO	1.6G	259*
s17	TO	160	118*	1.4K	MO	184*	1.6K	MO	201*	TO	1.6G	252*	1.6K	1.3G	**224**
s18	TO	160	226*	1.7	243	**226**	14	783	**226**	14	785	**226**	7	306	**226**
s19	TO	160	224*	2	244	**229**	14	774	**229**	14	776	**229**	7	306	**229**
s20	TO	160	213*	2.4	245	**233**	14	766	**233**	14	768	**233**	8	311	**233**
s21	TO	160	226*	1.7	244	**226**	14	783	**226**	14	785	**226**	7	306	**226**
s22	TO	160	222*	2	244	**229**	14	774	**229**	14	776	**229**	7	306	**229**
s23	TO	160	214*	2.4	245	**233**	14	766	**233**	14	768	**233**	8	311	**233**

no-part: no state partition **glh**,**dfs**: state prioritization
T: Time used (sec) **TO**: Time out (>2000 s)
M: Mem used (default Mb, $G \equiv$ Gb) **MO**: Mem out (>2 Gb)
d*: Peak depth before MO/TO **wd**: Witness Depth

TABLE II
STATISTICS OF HMC RUNS

Ex	glh				dfs			
	#S	#I	#F	#P	#S	#I	#F	#P
s1	1	17	0	2	0	16	1	0
s2	5	145	22	14	4	59	18	7
s3	3	106	10	13	4	90	18	11
s4	3	105	10	13	4	89	18	11
s5	3	108	22	14	4	133	27	16
s6	3	108	22	14	4	120	18	12
s7	5	145	22	14	4	118	18	12
s8	4	141	22	14	4	133	27	16
s9	5	145	22	14	4	69	18	9
s10	1	153	1	2	4	328	11	17
s11	1	157	1	2	4	331	11	17
s12	1	166	16	3	4	368	26	21
s13	7	646	58	58	4	368	26	21
s14	7	681	64	62	4	368	26	21
s15	1	166	16	3	4	368	26	21
s16	7	645	58	58	4	368	26	21
s17	2	516	29	10	4	332	11	17
s18	1	226	1	0	1	226	1	0
s19	1	229	1	0	1	229	1	0
s20	1	233	1	0	1	233	1	0
s21	1	226	1	0	1	226	1	0
s22	1	229	1	0	1	229	1	0
s23	1	233	1	0	1	233	1	0

#S: # of switches (PR→ST→PR)
#F: Peak size of frontier set F
#I: # of image computations
#P: # of calls to `Partition`

results show that the hybrid approach is more robust than applying both the state-based and stateless methods in isolation.

REFERENCES

[1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

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

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

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

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

[6] G. Holzmann. The model checker spin. *IEEE Trans. Software Engineering*, 1997.

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

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

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

[10] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. of Supercomputing*, 1991.

[11] T. Bultan, R. Gerber, and C. League. Composite model checking: verification with type-specific symbolic representations. In *ACM Trans. on Software Engineering Method*, 2000.

[12] Z. Yang, C. Wang, A. Gupta, and F. Ivančić. Mixed symbolic representations for model checking software programs. In *Formal Methods and Models for Co-Design*, 2006.

[13] C. Wang, Z. Yang, A. Gupta, and F. Ivančić. Using counterexamples for improving the precision of reachability computation with polyhedra. In *CAV*, 2007.

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

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

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

[17] H. Kim, F. Somenzi and H. Jin. Efficient Term-ITE Conversion for Satisfiability Modulo Theories. In *SAT*, 2009.

[18] F. Somenzi. CUDD: CU decision digram package. university of colorado at boulder. *ftp://vlsi.colorado.edu/pub/*.

[19] H. Cho, G. Hatchel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Trans. on CAD*, 1996.

[20] T. Yavuz-kahveci, M. Tuncer, and T.Bultan. A library for composite symbolic representations. In *TACAS*, 2001.

[21] M. Ganai and W. Li. Bang for the buck: Improvising and scheduling verification engines for effective resource utilization. In *Formal Methods and Models for Co-Design*, 2009.

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

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of PLDI*, 2005.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proc. of ICSE*, 2007.

[25] D. Yang and D. Dill. Validating and guided search of the state space. In *DAC*, 1998.

[26] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On Combining Formal and Informal Verification. In *CAV*, July 1997.

[27] S. Shyam and V. Bertacco. Distance-guided hybrid verification with guido. In *DATE*, 2006.

[28] A. Gupta, M. Ganai, C. Wang, and Z. Yang. Abstraction and bdds complement SAT-based bmc in diver. In *CAV*, 2003.

[29] K. McMillan. Interpolation and sat-based model checking. In *CAV*, 2003.

[30] G. Bischoff, K. Brace, G. Cabodi, S. Nocco, and S. Quer. Exploiting target enlargement and dynamic abstraction within mixed bdd and sat invariant checking. In *BMC workshop*, 2004.

[31] T. Arons, E. Elster, S. Ozer, J. Shalev, and E. Singerman. Efficient symbolic simulation of low level software. In *DATE*, 2008.

[32] M. Ganai and A. Gupta. Tunneling and Slicing: Towards Scalable BMC. In *DAC*, 2008.

[33] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *ICCAD*, 1997.

[34] S. Barner and I. Rabinovitz. Efficient Symbolic Model Checking of Software Using Partial Disjunctive Partitioning. In *Proc. of CHARME*, 2003.

[35] C. Wang, Z. Yang, F. Ivančić, and A. Gupta. Disjunctive image computation for embedded software verification. In *DATE*, 2006.

[36] D. Ward and F. Somenzi. Decomposing image computation for symbolic reachability analysis using control flow information. In *ICCAD*, 2006.