Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement

Chao Wang, Bing Li, HoonSang Jin, Gary D. Hachtel, *Life Fellow, IEEE*, and Fabio Somenzi

***Abstract*—The authors propose a scalable abstraction-refinement method for model checking invariant properties on large sequential circuits, which is based on fine-grain abstraction and simultaneous analysis of all abstract counterexamples of the shortest length. Abstraction efficiency is introduced to measure for a given abstraction-refinement algorithm how much of the concrete model is required to make the decision. The fully automatic techniques presented in this paper can efficiently reach or come near to the maximal abstraction efficiency. First, a fine-grain abstraction approach is given to keep the abstraction granularity small by breaking down large combinational logic cones with Boolean network variables (BNVs) and then treating both state variables and BNVs as atoms in abstraction. Second, a refinement algorithm is proposed based on an improved Ariadne's bundle[1] of synchronous onion rings on the abstract model, through which the transitions contain all shortest abstract counterexamples. The synchronous onion rings are exploited in two distinct ways to provide global guidance to the abstraction refinement process. The scalability of our algorithm is ensured in the sense that all the analysis and computation required in our refinement algorithm are conducted on the abstract model. Finally, we derive sequential don't cares from the invisible variables and use them to constrain the behavior of the abstract model. We conducted experimental comparisons of our new method with various existing techniques. The results show that our method outperforms other counterexample-guided methods in terms of both run time and abstraction efficiency.**

***Index Terms*—Abstraction refinement, binary decision diagram (BDD), formal verification, model checking, satisfiability (SAT).**

I. Introduction

THE PRIMARY obstacle to widespread use of formal-verification techniques, especially contemporary symbolic-model-checking algorithms [3], [4], remains the continuing explosive growth in the complexity of the model on which the verification property is specified. This is partly due to Moore's law—as the chips themselves grow in complexity, the size of the circuit assigned to one designer or design team grows commensurately. Another cause for the explosive growth is the increasing use of high-level hardware description languages (HDLs); models whose implementation requires thousands or tens of thousands of binary state variables (e.g., registers) may yet look modest when considering their HDL descriptions. Recent papers [5]–[16], including this one, have shown that symbolic-model checkers, extended to include an automated abstraction-refinement paradigm, hold great promise in dealing with state explosion.

Abstraction is an important technique to bridge the capacity gap between the model checker and large digital systems. When a system cannot be directly handled by the model checker, abstraction can be used to remove information that is irrelevant to the verification of the given property. Abstract interpretation, which can be regarded as a relation between the abstract system and the concrete system, was first used by Cousot and Cousot [17] in the context of static program analysis. There exist automatic abstraction techniques under which an entire class of properties is preserved. Bi-simulation-based reduction [18], [19], for instance, preserves the entire propositional μ-calculus. However, these property-preserving abstractions are often less attractive in practice because they are either hard to compute or do not achieve a drastic reduction [20]. A more practical approach is property-driven abstraction, which preserves or partially preserves only the property at hand. Balarin and Sangiovanni-Vincentelli [21], Long [22], and Cho *et al.* [23] have studied various ways of deriving an abstract model from the concrete system for model checking.

Abstraction refinement was introduced by Kurshan [24] in checking linear properties specified as ω-regular automata. In this paradigm, verification is viewed as an iterative process of synthesizing a simplified model that is sufficient to prove or refute the given property. The key issue here is to identify in advance which is relevant and which is not. In coordinated specification analysis (COSPAN) [25], the initial abstraction contains only the state variables in the property and leaves the other variables unconstrained. Since unconstrained variables can take arbitrary values, the abstract model is an overapproximation in the sense that it contains all possible execution traces of the original model, and possibly more. Therefore, when a linear time property holds in the abstract model, it also holds in the concrete model; when the property fails in the abstract model, the result is inconclusive. In case of an inconclusive result, the abstract model is refined by adding back some previously unconstrained variables. Note that such an overapproximate abstraction is applicable not only to safety properties but also to all universal properties, including linear temporal logic (LTL) [26], universal fragment of computation tree logic (ACTL), and ω-regular automata, because overapproximation suffices to prove these properties true.

For practical reasons, it is important to keep the abstraction process fully automatic. Manual abstraction can be very

Manuscript received July 18, 2005; revised September 30, 2005. Some of the preliminary results appeared in [1] and [2]. This paper was recommended by Associate Editor R. F. Damiano.

C. Wang is with the NEC Laboratories America, Princeton, NJ 08540 USA.

B. Li, G. D. Hachtel, and F. Somenzi are with the University of Colorado, Boulder, CO 80309 USA.

H. Jin is with Samsung Electronics, Korea.

Digital Object Identifier 10.1109/TCAD.2006.873897

[1]In the legend of Theseus, Ariadne's bundle contained one ball of thread to help Theseus navigate the labyrinth. In this paper, we work with multiple threads—hence, the "improved."

powerful when it is carried out carefully by experienced users. However, it often requires a significant amount of user's intervention and the in-depth knowledge of the design. In fact, manual abstraction is very labor intensive and can be error prone even for skilled users, which makes it hard for verification to keep up with the design schedule in real industry settings. Therefore, fully automated abstraction techniques are far more attractive in practice. In abstraction refinement, one typically starts with a coarse initial abstraction and then automatically augments the abstract model by iterative refinement.

The refinement schedule can be computed either statically, at the very beginning, or dynamically, at each refinement step. Algorithms in the first category include [27] and [28], where the refinement schedules are solely based on the structural information of the model, such as the pairwise latch relation and the variable-dependence graph. On the other hand, dynamically computed refinement schedules are guided by the model-checking result of the previous abstraction; therefore, they are generally more accurate than the statically computed ones. In Pardo's iterative μ-calculus model-checking procedure [29], refinement was driven by analyzing the sets of approximate satisfying states in the formula operation graph (which was improved later in [30]). For properties that have counterexamples, abstract counterexamples (ACEs) have been used to guide the refinement [5]–[7], [9], [31], [32]. Typically, counterexamples that appear in the abstract model but not in the concrete model are identified as spurious, and the goal of refinement is to get rid of the spurious counterexamples.

In counterexample guided dynamic-refinement methods, the symbolic analysis is often based on binary decision diagram (BDD) [33]. Yet another category is called proof-based abstraction refinement [10]–[16], which computes the dynamic-refinement schedules using a Boolean satisfiability (SAT) solver. Note that the methods in this category do not focus on identifying and removing spurious counterexamples. Instead, they rely on the SAT solver's capability of producing unsatisfiability cores. For an unsatisfying Boolean formula, the unsatisfiability core is a subset of the original Boolean formula, which is also unsatisfiable [34]. Given a model and an invariant property, the existence of finite-length counterexamples can be formulated as a Boolean SAT problem. When the problem is not satisfiable, the unsatisfiability core directly induces an abstract model that disables all counterexamples of that length.

The main challenge in abstraction refinement is related to the ability of generating a small final abstract model. The final or deciding abstraction is the one that can decide the truth of the given property. Since one can always start from a simple initial abstraction, the effectiveness of the refinement algorithm is critical in keeping the final abstract model small. The simplicity of the final abstract model is bounded ultimately by the degree of locality of the given property. In general, a high degree of locality is necessary for the success of the abstraction refinement. For a property whose proof or refutation relies on the detailed knowledge of the entire system, abstraction refinement is ineffective. In practice, however, user-specified properties often depend on only part of the system. This is largely due to the structured programming/design style adopted by engineers. Therefore, it is the refinement algorithm's responsibility to exploit fully the degree of locality of a given property. To measure the quality of different abstraction-refinement algorithms, we define abstraction efficiency as

$$\eta = \left(1 - \frac{\text{final abstract model size}}{\text{original model size}}\right).$$

For every pair of model M and property Φ, there exists an optimum, or maximum, abstraction efficiency η^*. Note that η^* is a property of the specific verification problem $\langle M, \Phi \rangle$, not a property of the abstraction-refinement algorithm. As a heuristic principle, the closer to the optimum value, the better a certain abstraction-refinement algorithm is.

Another important metric for abstraction refinement is its rate of convergence. This characterizes how quickly a refinement algorithm converges from the initial abstract model to a deciding abstraction. In practice, this can be measured by either the number of refinement iterations or the overall run time. We have observed cases for which some algorithms converge quickly to a near optimal abstraction while other algorithms spend a lot of time searching in vain for such an abstraction. In this paper, we propose a suite of algorithms that find, at each iteration, a set of refinement variables that are a subset of a near-optimal deciding abstraction.

A. Contribution

First, we propose a fine-grain abstraction approach to keep the abstraction granularity small by breaking down large combinational logic cones with Boolean network variables (BNVs) and then treating both state variables and BNVs as atoms in abstraction. It allows the refinement granularity to go beyond the usual "state variable level," which is in contrast with most abstraction methods in the prior art. In this method, the entire fan-in combinational logic cone of a state variable is either included in or completely excluded from the abstract model despite of the fact that not all these fan-in logic gates might be necessary for the verification even when the state variable itself is.

Second, we propose a data structure called the synchronous onion rings (SORs) to capture for an invariant property all the spurious counterexamples of the shortest length. We then give a new refinement algorithm called generational refinement of Ariadne's bundle (GRAB) to systematically analyze all the shortest counterexamples. GRAB has two novel features: First, it takes a generation of refinement steps to eliminate all spurious counterexamples supported by a given set of SORs. Second, each refinement in the current generation is chosen by a scalable game-based strategy whose computation depends solely on the current abstract model.

We also explore, in the context of abstraction refinement, the use of approximate reachable states of the invisible submodules to speed-up verification. We use approximate reachable states to disable certain valuations of the pseudoprimary inputs. Constraints from these neighboring invisible variables can prevent some spurious ACEs, therefore, leading to the decision of a property possibly earlier in the refinement cycle. To the best of our knowledge, we are the first to analyze these invisible submodules in the context of abstraction refinement; in

previous works, these invisible submodules were often discarded completely.

We have implemented our new algorithm in the model checker verification interacting with synthesis (VIS) [35], [36]. We present experimental comparisons of our algorithm to two recent counterexample-based refinement algorithms: the SepSet algorithm in [7] and the conflict analysis-based algorithm in [8], as well as the default BDD-based invariant-check algorithm in VIS, and the default-bounded model checker in VIS. For the purpose of experimental comparison, we have also implemented the algorithms of [7] and [8]. The experiments were conducted on circuits from both public domain and industry. Many of the models used were kindly provided by Chauhan *et al.* [8].

B. Related Work

Our fine-grain abstraction approach is unique in treating both state variables and BNVs as abstraction atoms. In fact, our refinement strategies must search in a two-dimensional space. Refinement in the sequential direction is comprised of state variables only, which is typical of much of the pioneering prior art of Clarke *et al.* [5], [7]. Refinement in the Boolean direction is comprised of BNVs only. Although the cut-set variables in the work of Wang *et al.* [6] and Glusman *et al.* [31] are similar to BNVs, they were not treated the same as state variables during refinement. We shall show that by carefully controlling the refinement direction between sequential and Boolean, we can produce significantly more concise refinement.

In [6], Wang *et al.* proposed a min-cut abstract model to replace the conventional coarse-grain abstract model. In the sequel, we use coarse grain when the smallest abstraction atom is a latch. They first defined a free-cut set of signals as those at the boundary of the transitive fan-in and transitive fan-out of the visible state variables. They then computed a min-cut set of signals between the combinational inputs and the free-cut signals; the logic gates above this min-cut were included in the reduced abstract model. However, the granularity is still at the state-variable level because logic gates above the free cut were always included in the abstract model, regardless of whether or not they were necessary for verification. In [8], Chauhan *et al.* adopted a similar approach, in which the further reduction of abstract model was achieved by prequantifying pseudoprimary inputs dynamically inside image computations. This approach shares the same drawback as that of Wang *et al.* [6].

In [31], Glusman *et al.* computed a min-cut set of signals between the boundary of the current abstract model and the combinational inputs and included logic gates above this cut in the abstract model. Since an arbitrary subset of the fan-in combinational logic gates of a state variable could be added, the abstraction granularity was at the gate level. However, there are some significant differences between our fine-grain abstraction and their method. First, fine-grain abstraction directly reduces the size of the transition relation by treating each elementary-transition relation as an abstraction atom, while their method aims at reducing the number of cut-set variables. Second, our refinement algorithm carefully controls the refinement direction, while theirs does not differentiate the two directions. Third, logic gates added by their method cannot be removed from the abstract model afterwards—even if later they are proved to be redundant, but the removal is possible in our method.

In [16], Zhang *et al.* proposed a technique called "dynamic abstraction," which maintains at different time steps separate visible variable subsets. Their approach can be viewed as a finer control of abstraction granularity over the time axis, since they are using different abstract models for computation at different time steps. However, at each time step, the abstraction atoms are still latches. Therefore, this approach is entirely orthogonal to our fine-grain abstraction.

A counterexample-guided refinement algorithm was proposed by Clarke *et al.* in [5], which relies on symbolically simulating a single ACE on the concrete model and then separating the deadend states from bad states. This algorithm was later improved in [7] by replacing the BDD-based concretization test with SAT and ILP solvers. Our refinement method differs from [5], [7], and other single counterexample-guided algorithms [8], [9] in that: 1) it handles all shortest ACEs rather than a single counterexample and 2) at each level in the concretization test, a set of abstract states, instead of just one abstract state, is used to constrain the bounded unrolled concrete model at each time step. We note that the number of spurious ACEs can be extremely large in practice, suggesting that the single counterexample-driven refinement is "a-needle-in-the-haystack" approach. To our knowledge, Clarke *et al.* [7] also had some preliminary experiments with multiple counterexamples and translation of multiple counterexamples to the SAT problem for invalidation; although, the work has not been published.

The refinement algorithm in [31] also relies on analyzing multiple counterexamples. In their approach, multiple abstract error traces are represented by a data structure called the multivalued counterexample. However, their multivalued counterexample does not guarantee to capture all the shortest ones, making it incapable of catching concretizable couterexamples at the earliest possible refinement step. Furthermore, their variable selection algorithm is based on the classification of invisible variables into strong 0/1 signals and conditional 0/1 signals. We shall show in Section V that strong 0/1 signals in particular are rare cases in practice. As a result, their refinement is often less accurate than GRAB.

In [32], Mang and Ho proposed a refinement algorithm based on controllability and cooperativeness analysis. Their cooperativeness analysis extracts a small subset of candidate input signals by applying a three-value simulation engine [6] to simulate the ACEs and then ranking all the inputs (i.e., invisible state variables and BNVs) according to various criteria. Their controllability analysis is independent of any particular counterexample (i.e., based on SORs); it is applied to a subset of input signals by scoring them according to a game-theoretic formula derived from the SORs. These two proposed analyses are then carefully integrated together to better refine the abstract model. Their controllability analysis is an improvement of our GRAB algorithm. Their experimental results showed a significant improvement over both GRAB and the RFN method in [6].

The proof-based abstraction-refinement methods in [10]–[16] also handle implicitly all the counterexamples of a finite length. These methods differ from ours in that their refinement variable-selection algorithms are all SAT-based, i.e., relying on the SAT solver's capability to produce succinct unsatisfiability proofs. In contrast, our core refinement variable-selection algorithm is pure BDD-based, even though we use SAT as well in predicting the refinement direction and in the concretization test. We note that a small unsatisfiability proof, i.e., the one with a small subset of Boolean variables or clauses, does not automatically give a small refinement set [13], [37]. Both proof-based and counterexample-based methods have their own advantages and disadvantages. A detailed experimental comparison of GRAB with a proof-based refinement algorithm can be found in our recent paper [14], which shows that these two kinds of methods complement each other on the various test cases. Amla *et al.* [38] also published results of their experimental evaluation of the various SAT-based abstraction methods. There is also a trend of combining counterexample-based methods and proof-based methods in abstraction refinement [39].

C. Organization of Paper

After establishing notation in Section II, we present our fine-grain abstraction approach in Section III. We then discuss in Section IV how to compute and deploy Ariadne's bundle in the abstraction refinement process. This leads to a discussion of the new refinement variable-selection algorithm in Section V. We explain in Section VI our SAT-based methods for predicting the refinement direction and minimizing the refinement set. We then give experimental results in Section VII and conclude in Section VIII.

II. Preliminaries

In symbolic model checking, one manipulates sets of states instead of each individual state. Both the transition relation of the model and the sets of states are represented by Boolean functions called characteristic functions, which can in turn be represented by BDDs or Boolean formulas. Let the model be given in terms of:

1) a set of present-state variables $x = \{x_1, \ldots, x_m\}$;
2) a set of input variables $w = \{w_1, \ldots, w_n\}$;
3) a set of next-state variables $y = \{y_1, \ldots, y_m\}$.

The state transition graph of the model can be represented symbolically by $\langle T, I\rangle$, where $T(x, w, y)$ is the characteristic function of the transition relation, and $I(x)$ is the set of initial states. A state is a valuation of either the present-state or next-state variables. If the valuation of the state variables makes the initial predicate $I(x)$ true, the corresponding state is an initial state. Let $\tilde{x}$, $\tilde{y}$, and $\tilde{w}$ be the valuations of x, y, and w, respectively, then the transition relation $T(\tilde{x}, \tilde{w}, \tilde{y})$ is `true` if and only if under the input condition $\tilde{w}$ there is a valid transition from State $\tilde{x}$ to State $\tilde{y}$.

Computing the image or preimage is the most fundamental step in symbolic model checking. The image of a set of states consists of all the successors of these states in the state transition graph; the preimage of a set of states consists of all their predecessors. In this paper, we use the two operators $\mathsf{EY}(D)$ and $\mathsf{EX}(D)$ to represent the image and preimage of D under the transition relation T. These two basic operations are formally defined as follows:

$$\mathsf{EY}_T(D) = \{s' | \exists s \in D : (s, s') \in T\}$$
$$\mathsf{EX}_T(D) = \{s | \exists s' \in D : (s, s') \in T\}.$$

When the context is clear, we will drop the subscripts and use EX and EY instead. Given the symbolic representation of the transition relation T and the state set D, the image and preimage are computed as

$$\mathsf{EX}_T(D) = \exists y, w \,.\, T(x, w, y) \wedge D(y)$$
$$\mathsf{EY}_T(D) = \exists x, w \,.\, T(x, w, y) \wedge D(x).$$

In this paper, we are focusing on invariant properties of the form $\mathsf{AG}p$; that is, the predicate p holds globally. Many interesting temporal logic properties, including $\mathsf{AG}p$, can be evaluated by applying EX and EY repeatedly until a fix point is reached [40]. The set of states that are reachable from I, for instance, can be computed by the least fix-point computation

$$\mathsf{EP}I = \mu Z \cdot I \cup \mathsf{EY}(Z).$$

This computation is called the forward reachability analysis—one that repeatedly computes the image of the set of already-reached states starting from the initial states I until it stops growing. Similarly, the set of states from which D is reachable, denoted by $\mathsf{EF}D$, can be computed by the least fix-point computation

$$\mathsf{EF}D = \mu Z \cdot D \cup \mathsf{EX}(Z).$$

This is often called the backward reachability analysis.

Invariant properties can be decided by the reachability analysis. Let P denote the set of states labeled p, then $\mathsf{EP}I \subseteq P$ means that the property $\mathsf{AG}p$ holds. If the property fails, the model checker can produce a counterexample, i.e., a sequence of states $(s_0, s_1, \ldots, s_k)$ such that $s_0 \in I$ and $s_k \models \neg p$.

The set of reachable states can be computed by BDD-based symbolic fix-point computation, in which the states and transition relation are represented by BDDs, while set operations including intersection, union, and existential quantification are implemented as BDD operations. Symbolic model checking [4] was a major breakthrough in boosting the capacity of model-checker techniques leading to the subsequently widespread acceptance of model checking in hardware verification. When BDDs are used as the underlying data structure, the complexity of symbolic model checking depends on the size of the BDDs that represent the operands. Because of this, the search for heuristics to avoid the BDD blowup has been one of the major research areas in formal verification.

III. Fine-Grain Abstraction

The model is considered as a formal description of the behavior of the system. Abstraction preserves only the part of

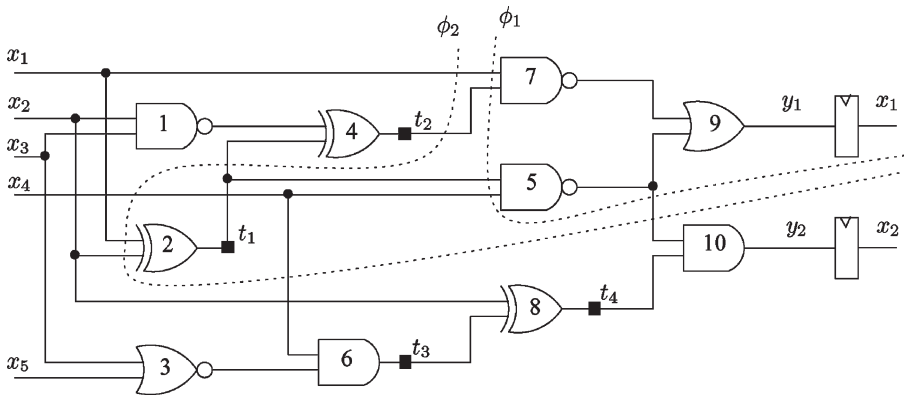


Fig. 1. Illustration of fine-grain abstraction.

behavior that is relevant to the verification of the given property, in the hope that the simplified model is easier to verify. For digital circuits, the abstract models can be constructed directly from a high-level description of the system, even before the concrete model of the system is available.

A. Defining Abstract Model

We consider the concrete model $\mathcal{M} = \langle T, I \rangle$ to be the synchronous composition of many submodules. The transition relation T is then the conjunction of individual transition subrelations. In the simplest form, every state variable together with its bit transition relation is considered as a submodule. Let $J = \{1, \ldots, m\}$ be the indexes of the state variables; then

$$T(x, w, y) = \bigwedge_{j \in J} T_j(x, w, y_j)$$

where $T_j(x, w, y_j)$ is the bit transition relation of the jth binary state variable y_j and, thus, depends on one next-state variable. If y_i has a transition function Δ_i, then the transition relation for y_i can be formulated as $T_j = (y_j \leftrightarrow \Delta_j(x, w))$.

Any overapproximation of T and I, denoted by $\widehat{T}$ and $\widehat{I}$, respectively, induces an abstract model. A widely adopted way of overapproximating T is replacing some of the bit relation T_j by tautology, since tautology does not impose any constraint on how the transitions can be made. Note that T_j is treated as an atom for abstraction in this approach, since it is either included in or excluded completely from the abstract model.

Assume that the abstract model contains a subset of state variables $\widehat{J} = \{1, \ldots, k\} \subseteq J$. Let $\hat{x} \subseteq x$ be the subset of present-state variables and $\hat{y} \subseteq y$ be the subset of next state variables, then $\widehat{T}$ is defined as follows:

$$\widehat{T}(x, w, \hat{y}) = \bigwedge_{j \in \widehat{J}} T_j\left(\{\hat{x}, \breve{x}\}, w, y_j\right).$$

Variables in $\hat{x}$ are called the visible state variables, and variables in $\breve{x} = x \setminus \hat{x}$ are called the invisible state variables. Note that although $\widehat{T}$ has a subset $\hat{y}$ of the next-state variables, it may contain some invisible present-state variables $\breve{x}$; these invisible present-state variables are treated as inputs (also called pseudo-primary inputs). The set of initial states $\widehat{I}(\hat{x})$ is an existential projection of $I(x)$: An abstract state is initial if and only if it contains a concrete initial state.

The abstract model $\widehat{A} = \langle \widehat{T}, \widehat{I} \rangle$ is defined in exactly the same concrete state space, only with more transitions among the states and possibly more states labeled as initial. The simplification of the abstract model is not in the size of the state transition graph, but mainly in the size of the BDD representation of the transition relation. Due to less number of conjuncts, the characteristic function can be represented by a simpler Boolean formula. This interpretation of abstraction appears natural when analyzing symbolic-graph algorithms.

Restricting the abstraction granularity at the state-variable level is not suitable for verifying industrial-scale systems with extremely large combinational logic cones. At this level, T_j can be either $(y \leftrightarrow \Delta_j)$ or tautology, depending on whether the corresponding state variable is included or not, but not an arbitrary Boolean function in between. However, not all logic gates in the combinational logic cone might be necessary for the verification, even if the state variable itself is. Unnecessarily including the irrelevant information can make the abstract model too complex for the model checker to deal with. Let $\leq$ denote the subset or equal relation, then an abstraction $\widehat{T}_j$ such that $(y \leftrightarrow \Delta_j) \leq \widehat{T}_j \leq \mathbf{1}$ is often more appropriate; however, this is not possible under the "coarse-grain" abstraction approach.

Based on this observation, we propose a fine-grain abstraction approach to go beyond the state-variable level. It considers not only the state variables but also the BNVs. BNVs are intermediate variables inserted into the combinational logic cones of latches to partition large cones so that a compact BDD representation of their transition relations is possible. Once inserted, each BNV is associated with a small area of the combinational circuit; similar to the state variables, there is an elementary-transition relation for each BNV. The transition relation of the entire system is the conjunction of all these elementary-transition relations.

The following example shows how fine-grain abstraction works. In Fig. 1, there are ten gates in the fan-in combinational logic cones of the two latches. Here, y_1 and y_2 are the next-state variables, and $x_1, \ldots, x_5$ are the present-state variables, among which x_1 and x_2 correspond to y_1 and y_2. Let Δ_{y_1} be the output function of Gate 9 in terms of the present-state variables and inputs only; similarly, let Δ_{y_2} be the output function of Gate 10. Δ_{y_1} and Δ_{y_2} are also called the transition functions of

Latch 1 and Latch 2, respectively. According to the definition in the previous section, the bit transition relation of Latch 1 is defined as

$$T_1 = y_1 \leftrightarrow \Delta_{y_1}(x_1, x_2, x_3, x_4).$$

We now insert new variables $t_1, t_2, t_3,$ and t_4 to partition the fan-in combinational cones of the two latches. We use δ_v to represent the output function of the signal v, in terms of both state variables as well as newly added BNVs. This is in contrast to Δ_v, which is in terms of the state variables only. These new functions and their corresponding finer grain elementary-transition relations are defined as

$$\begin{aligned}
\delta_{t_1} &= x_1 \oplus x_2 & T_{t_1} &= t_1 \leftrightarrow \delta_{t_1} \\
\delta_{t_2} &= \neg(x_2 \wedge x_3) \oplus t_1 & T_{t_2} &= t_2 \leftrightarrow \delta_{t_2} \\
\delta_{t_3} &= \neg(x_3 \vee x_5) \wedge x_4 & T_{t_3} &= t_3 \leftrightarrow \delta_{t_3} \\
\delta_{t_4} &= x_2 \oplus t_3 & T_{t_4} &= t_4 \leftrightarrow \delta_{t_4} \\
\delta_{y_1} &= \neg(x_1 \wedge t_2) \vee \neg(x_4 \wedge t_1) & T_{y_1} &= y_1 \leftrightarrow \delta_{y_1} \\
\delta_{y_2} &= \neg(x_4 \wedge t_1) \wedge t_4 & T_{y_2} &= y_2 \leftrightarrow \delta_{y_2}.
\end{aligned}$$

Note that the state variable y_1 is now associated with δ_{y_1} and T_{y_1} only. The bit transition relation of Latch 1 is a conjunction of three elementary-transition relations

$$T_1 = T_{y_1} \wedge T_{t_1} \wedge T_{t_2}.$$

In coarse-grain abstraction methods where only state variables are treated as atoms, when Latch 1 is included in the abstract model, all the six fan-in gates (Gate 1, 2, 4, 5, 7, and 9) are also included; that is, $\widehat{T} = T_{y_1} \wedge T_{t_1} \wedge T_{t_2}$. In fine-grain abstraction, when Latch 1 is in the abstraction, only those gates covered by the elementary-transition relation T_{y_1} are included; this is indicated in the figure by the cut ϕ_1, which contains Gates 5, 7, and 9. In the successive refinement steps, only the clusters that are relevant to verification are added. In the next section, we will present an algorithm to identify which clusters should be included. Meanwhile, let us assume that the current abstract model is not sufficient and the BNV t_1 is added during refinement. This is indicated by the new cut ϕ_2 in the figure, which means $\widehat{T} = T_{y_1} \wedge T_{t_1}$. The abstract model now contains Latch 1 and Gates 2, 5, 7, and 9. Continuing this process, it may add $y_2, t_4, \ldots$ until a proof or refutation is found.

It is possible in the fine-grain approach that gates covered by the transition cluster T_{t_2} (i.e., Gates 1 and 4) never appear in the abstract model if the gates are indeed irrelevant to the verification of the given property. This demonstrates the advantage of fine-grain abstraction. In a couple of industry strength circuits, including one from the PicoJava Instruction Unit, we have observed that over 90% of the gates in some large fan-in cones are indeed redundant, even though the corresponding latches are necessary.

Our BNV-based approach can have a finer granularity of abstraction than the min-cut-based approach in [6]. Take Fig. 1 as an example, according to [6], where the free-cut set is $\{x_1, x_2, x_3, x_4, t_3\}$, which means that the free-cut model already includes everything except Gates 3 and 6. Since this free-cut set happens to be a min-cut set as well (another min-cut set is $\{x_1, x_2, x_3, x_4, x_5\}$), this free-cut model is also their min-cut abstract model.

In fine-grain abstraction, the granularity depends on the size of the elementary-transition relations as well as the algorithm used to perform the partition. In the current implementation, we apply the frontier [41] algorithm to selectively insert BNVs. We choose the frontier algorithm because it is a stable technique for reducing the sizes of partitioned BDDs; however, some kind of structural analysis may further improve the quality of the inserted BNVs. The frontier algorithm was initially proposed in the context of symbolic image computation. It works as follows: First, the elementary-transition function of each gate is computed from the combinational inputs to the combinational outputs, in some topological order. If the BDD size of an elementary-transition function exceeds a given threshold, a new BDD variable is inserted. For all the gates in the fan outs of that gate, their elementary-transition functions are computed in terms of the new variable. Each BNV or state variable is then associated with a BDD δ_{t_k} or δ_{y_j} for describing its transition function.

The actual abstraction granularity can be fine tuned by controlling the BDD threshold of the frontier partitioning algorithm. In the extreme case, the BDD threshold can be set to one, i.e., a BNV is created for every logic gate. In that case, the optimum abstraction, among all the possible final abstract models, is the one that has the fewest logic gates. In some sense, one can establish a connection between abstraction refinement and logic synthesis. Abstraction refinement is an iterative process of synthesizing a small abstract model that can prove or refute the given property.

We now formally define the fine-grain abstract model. We denote the set of BNVs by $t = \{t_1, \ldots, t_n\}$. Every variable t_k is associated with a transition relation $T_{t_k} = (t_k \leftrightarrow \delta_{t_k})$, and every state variable is associated with a fine-grain transition relation $T_{y_j} = (y_j \leftrightarrow \delta_{y_j})$. Let $J = \{1, \ldots, m\}$ and $K = \{1, \ldots, n\}$, then the concrete transition relation becomes

$$T(x, w, t, y) = \bigwedge_{j \in J} T_{y_j}(x, w, t, y_j) \wedge \bigwedge_{k \in K} T_{t_k}(x, w, t, t_k).$$

Assume that our refinement algorithm (in Section IV) returns a fine-grain abstract model consisting of $m' \leq m$ state variables and $n' \leq n$ BNVs, i.e., $\widehat{J} = \{1, \ldots, m'\} \subseteq J$ and $\widehat{K} = \{1, \ldots, n'\} \subseteq K$, then

$$\widehat{T} = \bigwedge_{j \in \widehat{J}} T_{y_j}\left(\hat{x}, \{\check{x}, w\}, \hat{t}, y_j\right) \wedge \bigwedge_{k \in \widehat{K}} T_{t_j}\left(\hat{x}, \{\check{x}, w\}, \hat{t}, t_k\right).$$

Here, $\hat{x} = \{x_j | j \in \widehat{J}\}$ is the subset of visible present-state variables, $\hat{y} = \{y_j | j \in \widehat{J}\}$ is the subset of visible next-state variables, and $\hat{t} = \{t_k | k \in \widehat{K}\}$ is the subset of BNVs in the abstract model. All the remaining (invisible) present-state variables and BNVs go into $\check{x}$. During symbolic model checking, variables in $\check{x}$ are treated as inputs or pseudoprimary inputs.

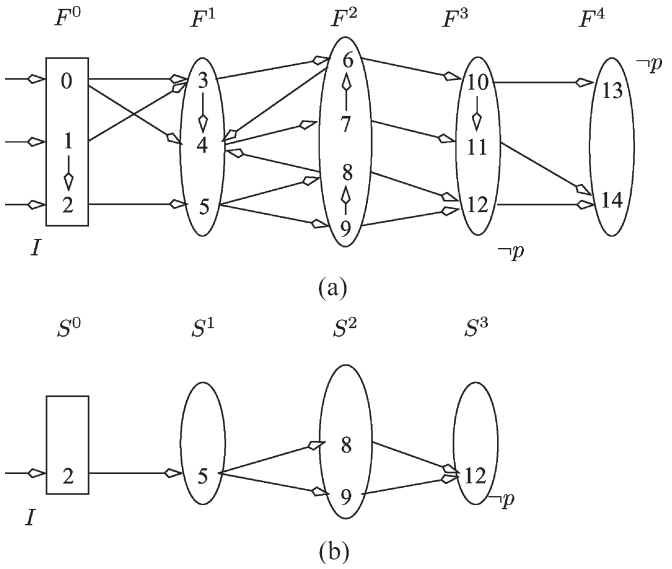


Fig. 2. Ariadne's bundle of SORs. (a) Current abstraction. (b) SORs.

B. SORs

Counterexamples found in the abstract model may not be real paths because some transitions that are responsible for the counterexamples may be forbidden in the concrete model. Checking whether these ACEs are real or not is called the concretization test. Conceptually, concretization test can be done by reconstructing the abstract paths in the concrete model. If they cannot be reconstructed, the counterexamples are spurious. Refinement relies on the analysis of the spurious counterexamples.

The number of counterexamples to a general LTL property can be infinite (e.g., when the counterexamples contain cycles). Even if one focuses on invariant properties and on counterexamples of the shortest length, the number of counterexamples can still be extremely large. We have observed in practice, for instance, that the number of shortest counterexamples to an invariant property is 10^{45} in a model with 100 state variables. This suggests that analyzing them one by one through enumeration is not a good strategy. Since it is not clear yet how to identify the more representative ones, arbitrarily choosing one counterexample to guide the refinement is also "a-needle-in-the-haystack" approach. It is desirable to capture as many counterexamples as possible and analyze them simultaneously. For safety properties, one can actually capture symbolically all the counterexamples of the shortest length.

We propose a new data structure called the SORs to capture all the shortest counterexamples of an invariant property. Consider the property AGp as an example. The state transition graph of the abstract model is shown in of Fig. 2(a). Forward reachability analysis from I gives the forward onion rings, which are the sets of new states encountered at each iteration during the breadth-first search. $F^1 = \{3, 4, 5\}$, for instance, is the set of states that can be reached in one step from the initial states. Note that the $\neg p$ states are first reached at the third step of the search. An analogous backward reachability analysis from the $\neg p$ states in F^3 would reach states $\{8, 9\}$ in F^2, $\{5\}$ in F^1, and the initial state 2. Once the forward and backward reachable onion rings are available, the intersection of the two sets of states at each step gives the SORs

$$S^0 = \{2\},\ S^1 = \{5\},\ S^2 = \{8, 9\},\ S^3 = \{12\}.$$

Note that both forward and backward onion rings are standard representation for counterexamples in BDD-based symbolic model checkers; they have been used also in a bounded model checking (BMC) induction proof to restrict the SAT search space [42].

The term "Ariadne's bundle" is used to denote the subrelation T^B induced by considering only the transitions between a state at one step to another state in the next step in the SORs. It comprises the bundle of all shortest counterexamples and no other counterexample.[2] Note that the state–transition graph of the Ariadne's bundle has significantly less states and transitions than the abstract model. In this simple case, there are two shortest counterexamples of length three. In practice, however, the number of counterexamples in the SORs is typically large.

C. Multithread Concretization Test

Concretization test of the ACEs cannot be accomplished by simulation, even if a single abstract path is to be reconstructed. This is because an abstract path may not have a complete set of assignments to all the input variables of the concrete model—one abstract transition often corresponds to multiple concrete transitions. Various symbolic techniques have been proposed for concretization test. In [5], BDD-based image computation was applied in the concrete model to reconstruct all the concrete paths inside a single abstract path. In [6], the search of a concrete path inside a single abstract path was performed by an automatic test pattern generation (ATPG) engine. In [7] and [8] the reconstruction was performed by SAT solvers. However, one thing common to all these methods is that the concretization test deals with only a single-ACE.

The problem here is the simultaneous concretization of all the shortest counterexamples. We solve this multithread concretization test by formulating it as a Boolean satisfiability problem and then applying an SAT solver. Given the length L SORs $\{S^0, \ldots, S^L\}$, we define the SAT problem as $\Psi = \Psi_A \wedge \Psi_S$

$$\Psi_A = I(V^0) \wedge \bigwedge_{0 \le i < L} T(V^i, U^i, V^{i+1})$$
$$\Psi_S = \bigwedge_{0 \le i \le L} S^i(V^i)$$

where Ψ_A represents the unrolling of the concrete transition relation for L time frames, and Ψ_S represents the constraints coming from the abstract SORs (Fig. 3). We use V^i to represent the state variables at the ith time frame and U^i to represent the internal nodes and inputs. The predicate $I(V^0)$ represents that all states at time frame zero are initial, while $T(V^i, U^i, V^{i+1})$

[2]There is an interesting analogy between the abstract counterexamples and the magic Ariadne's thread. In the Greek mythology, Theseus needs the thread to navigate through the Labyrinthus to kill the monster Minotaurus; in abstraction refinement, one needs the guidance of abstract counterexamples to remove the "false negatives."

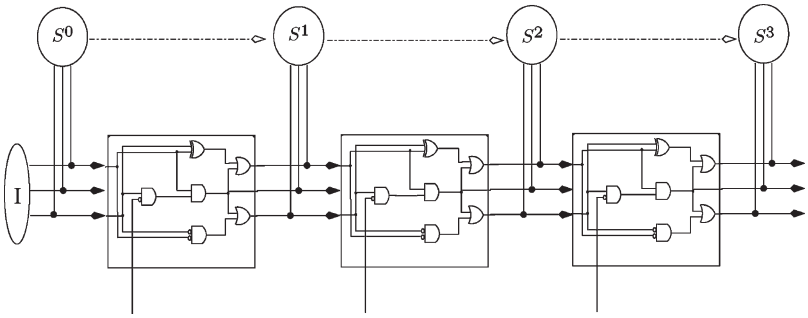


Fig. 3. Multithread concretization test by constraining BMC instance with SORs.

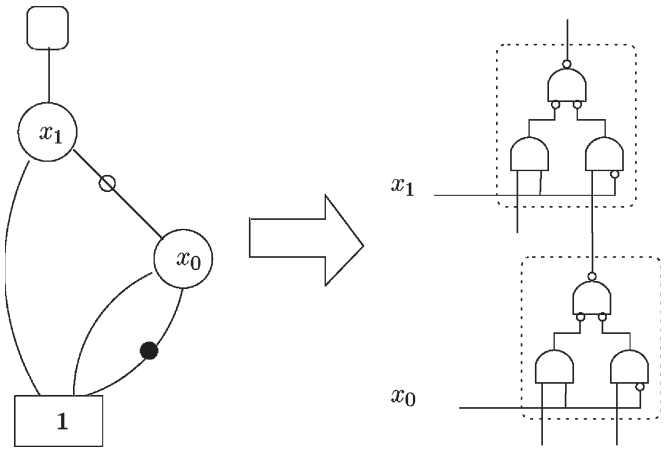


Fig. 4. Translating BDD into combinational circuit.

indicates that transitions between time frames i and $i+1$ must obey the concrete transition relation. The predicate $S^i(V^i)$ restricts states at the ith time frame into the ith ring of the SORs.

Formula Ψ is satisfiable if and only if there exists a concrete counterexample (CCE) inside the abstract SORs. When Ψ is satisfiable, the set of assignments returned by the SAT solver represents a concrete path from an initial state to a $\neg p$ state.

In order to decide Ψ with an SAT solver, we need to put the formula into the conjunctive normal form (CNF). In particular, we need to translate the rings $\{S^i\}$ into CNF because the symbolic model checker normally returns S^i as a BDD. The translation from a BDD to a Boolean circuit is illustrated in Fig. 4. The basic idea is to translate each BDD node into a two-to-one multiplexers. Note that each BDD node represents an If-Then-Else (ITE) branching condition. If the current variable is true, the output node of the BDD is the same as the left child, otherwise, the right child. ITE nodes can be mapped directly to multiplexers. Since each multiplexer consists of 3 and gates, the Boolean circuit is linear in the number of BDD nodes. This is the translation scheme we implemented in our framework. It was first used by Gupta *et al.* [11] in speeding up BMC with BDD learning. This translation scheme requires the addition of a large number of auxiliary variables to encode the internal logic gates. An alternative approach is to enumerate all the minterms of the complemented function $\neg S^i$, and convert the minterms into CNF clauses. A minterm of the Boolean function corresponds to a root-to-leaf path in its BDD. Although no auxiliary variable is required, the number of root-to-leaf paths can still be exponential in the number of BDD nodes. In [43], a hybrid encoding scheme was proposed to make a tradeoff between the above two approaches.

IV. GRAB

In this section, we illustrate the generic process of abstraction and refinement by treating the simple example in Fig. 5. We then give the overall algorithm of our generational refinement based on Ariadne's bundle.

A. Example

The concrete model $\mathcal{M}$ in Fig. 5 is the synchronous composition of three submodules: $M_1, M_2,$ and M_3. That is

$$\mathcal{M} = M_1 \parallel M_2 \parallel M_3.$$

Each component M_i has one state variable v_i. The state variable v_1 can take four values and thus must be implemented by two binary variables; the other two state variables (v_2, v_3) are binary variables. The variable v_4, which appears on the edges in M_1, is a primary input. The property of interest is $\mathsf{AG}(v_1 \neq 3)$, i.e., State 3 in M_1 is not reachable. The right part in Fig. 5 gives the state-transition graph of the concrete model. It is clear that the property fails in the concrete model, as shown by the bold edges, which exhibit a CCE of length 4: (000, 111, 200, 211, 300).

The initial abstraction is $\widehat{\mathcal{A}} = M_1$, in which only the state variable appearing in the given property is preserved, and all the other state variables are treated as pseudoprimary inputs. There is an ACE of length 3: $(0\_\_, 1\_\_, 2\_\_, 3\_\_)$. This ACE is spurious because it is not concretizable; no direct transition is possible from 200 to $3\_\_$ in $\mathcal{M}$.

Although this example is simple, it demonstrates an important aspect of the abstraction refinement process. Refinement algorithms based on separating deadend states from bad states, like those in [5], [7] since they are actuated by a single counterexample, may pick only variable v_2 for refinement. In fact, v_2 can separate the bad states {211, 210} from the deadend state {200}. Here, {200} is a deadend state (cf., [5]) since it is reachable from the initial state but does not have a concrete transition to $\{3\_\_\}$, while {211, 210} are bad states since they are not reachable from the initial state but have concrete transitions to $\{3\_\_\}$. However, after this refinement, an ACE of the same length still exists—it can be $(00\_, 10\_, 21\_, 30\_)$. Therefore, $\{v_2\}$ is not a sufficient refinement set to kill the

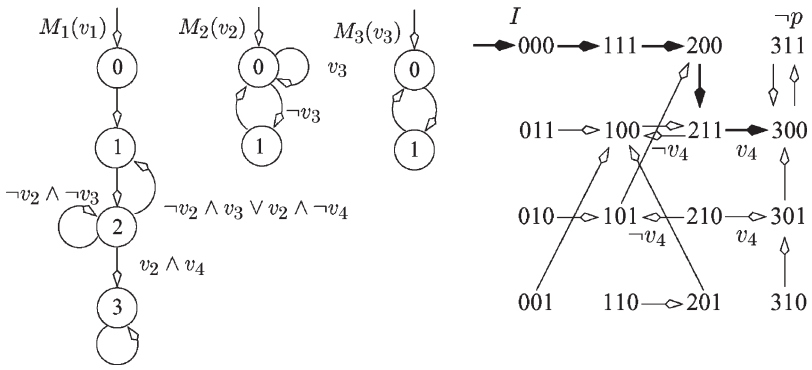


Fig. 5. Example for abstraction refinement.

ACE $(0\_\_, 1\_\_, 2\_\_, 3\_\_)$. This is suggestive of the danger of placing too much refinement emphasis on a single ACE. Of course, it is much more of a problem when the SORs contain an extremely large number of counterexamples. In this case, choosing the refinement variables based on a single counterexample can be ineffective.

We now illustrate the SOR-based generational refinement framework using the above example. In building the SORs, self-loops and back edges are pruned away to focus on the shortest counterexamples in the current abstract model. When $\widehat{\mathcal{A}} = M_1$, the SORs contain just the four states of M_1, which are connected by the four forward edges. Since the first generation of shortest ACEs are of length 3, the refinement process starts by dealing with the SORs of length three until all paths in them are killed. The initial SORs are shown in Fig. 6(a). Note that in M_1 only edges from State 2 to States 1, 2, and 3 are labeled. As will be discussed in Section V, these labels cause the variable selection routine to pick variable v_2 for the first refinement. The refined abstract model is $\widehat{\mathcal{A}} = M_1 \parallel M_2$; however, $\widehat{\mathcal{A}}$ is not constructed in this naive way, but by a more efficient two-step process.

First, we split the states according to the labels on their outgoing edges, as shown in Fig. 6(b). Because of the label $v_2 \wedge v_4$, the last abstract edge $(2\_\_, 3\_\_)$ is split into two rather than four refined edges. State $20\_$ is now backward unreachable from $\neg p$, so the two incoming edges, $(10\_, 20\_)$ and $(11\_, 20\_)$, are removed. The outgoing edges from State $01\_$ are removed as well because State $01\_$ is not an initial state. States like $20\_$ are called the deadend states. The concept of deadend states is critically involved in the refinement variable selection algorithm, as discussed below in Section V. Note that all the splits that make the SORs change from the one in Fig. 6(a) to the one in Fig. 6(b) are done before M_2 is brought in. The second step is to actually take the composition of M_2 with the remaining edges of the SORs. This kills the edges $(11\_, 21\_)$ and $(21\_, 31\_)$ and leads to the reduced SORs in Fig. 6(c).

After the above refinement step, the number of length-3 spurious counterexamples is decreased but they have not been removed completely. At this point, v_3 will be selected as the next refinement variable. We then proceed to again take the first part of the two-step refinement process, as illustrated in Fig. 6(d). The result is a disconnection of I and $\neg p$ because there is no outgoing edge from the sole remaining initial state. At this

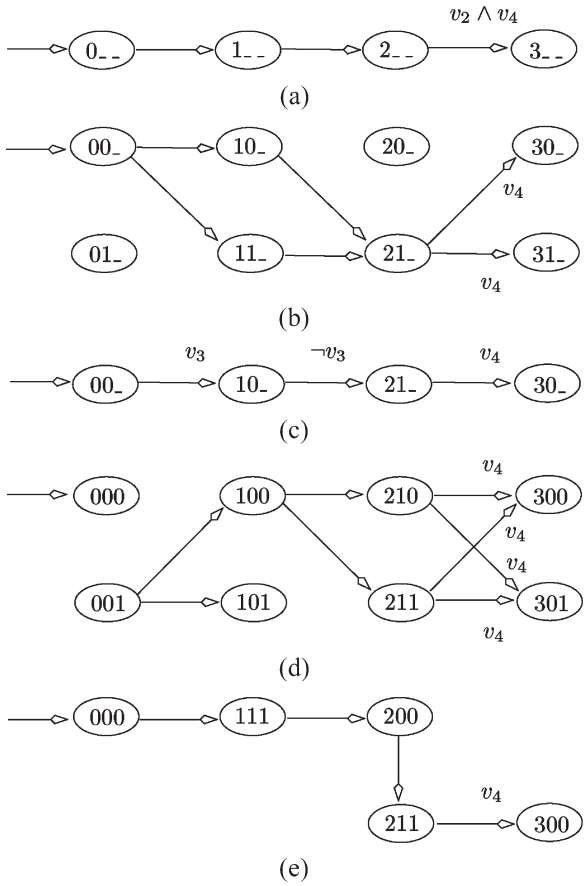


Fig. 6. Generational refinement process. (a) Original SORs. (b) Split states. (c) Account for TR of M_2. (d) Split states again; and (e) compute length-4 SORs.

point, it has been proved that no CCE of length 3 exists, so this generation of refinements is complete. Note that during the two refinements in the length-3 generation (i.e., adding v_2 and adding v_3), the SORs are updated incrementally inside previous SORs. The BDD don't cares associated with this incremental process lend critical efficiency to SOR's refinement process.

Next, the SORs are built from scratch, which now are of length 4, as is shown in Fig. 6(e). This final set of SORs contains a single counterexample that is concretizable in $\mathcal{M}$, as discussed above in reference to Fig. 5. Therefore, the given property fails.

The proposed refinement algorithm does not try to remove all the spurious ACEs in one shot. Instead, it identifies local

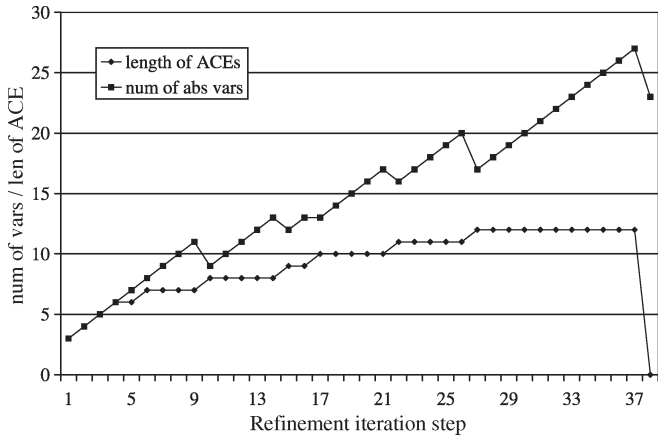


Fig. 7. Effect of generational refinement with refinement minimization.

variables that are critical to refinement by exploiting the global guidance provided by the SORs. It may take a set of refinement steps, called a generation of refinements, to remove all the shortest ACEs of a given length.

The effect of generational refinement is illustrated in Fig. 7. The data are obtained from a circuit design called D20 in which the given invariant property is true. The upper curve represents the number of state variables in the abstract model at different refinement steps, and the lower curve is the length of the shortest ACE. A generation consists of a number of consecutive refinement steps, all with SORs of the same length. Note that every time the length of the shortest ACE changes, the number of abstract variables may decrease; this is due to the greedy refinement-minimization procedure that tries to keep the abstraction as small as possible by removing redundant variables. We will explain refinement minimization in Section VI-B. Experience shows that this greedy minimization is critical in achieving a high abstraction efficiency.

B. Overall Algorithm

Let $\{S^0, S^1, \ldots, S^L\}$ be the length-L synchronized onion rings, where S^0 is a subset of initial states, S^L is a subset of states satisfying $\neg p$, and S^j is a set of states on the shortest abstract paths from S^0 to S^L. The pseudo code of the abstraction-refinement algorithm is given in Fig. 8. It is called GRAB for generational refinement of Ariadne's bundle. GRAB accepts as inputs the concrete model $\mathcal{M}$ and the property Φ (of the form $\Phi = \mathsf{AG}p$).

The initial abstract model $\widehat{\mathcal{A}}$ contains only those state variables that appear in the local support of the property, but no BNV. The outer loop is over the length L of the current generation of SORs. With the abstract model being gradually refined, L is guaranteed to grow monotonically in the outer loop. The action starts in Line 3, where BDD-based forward reachability analysis is used to compute the forward onion rings from the initial states to $\neg p$ states. If $\neg p$ cannot be reached in $\widehat{\mathcal{A}}$, it cannot be reached in $\mathcal{A}$ either. In this case of early termination, GRAB returns `true`. Otherwise, the first set of SORs is computed. An SAT-based concretization test is then performed in the concrete model. Here, it simultaneously tries to concretize all the ACEs in the SORs by one satisfiability

```
    GRAB(M, Φ)      {
1       Â = INITIALABSTRACTION(M, Φ);
2       while (true) {    //Loop over SORs with different length
3           {S^l} = COMPUTESORS(Â, Φ);
4           if ( {S^l} is empty )
5               return true;
6           CCE = MULTITHREADCONCRETIZATION(M, {S^l});
7           if (CCE not empty)
8               return (false, CCE);
9           {S_R^l} = {S^l};
10          while (true) {    //Loop at the current length
11              Â = REFINEABSTRACTION(Â, {S_R^l});
12              {S_R^l} = REDUCESORS(Â, {S_R^l});
13              if ({S_R^l} is empty)
14                  break ;
15          }
16          Â = REFINEMENTMINIMIZATION(Â, {S^l});
    }   }

    REFINEABSTRACTION(Â, {S^l})      {
17      w_S = { }, w_E = ŵ;
18      while (|w_S| < threshold) {
19          v = PICKBESTVAR(Â, {S^l});
20          w_S = w_S ∪ {v}, w_E = w_E \ {v};
21      }
22      return COMPUTEABSTRACTION(Â, w_S) ;
    }
```

Fig. 8. Abstraction-refinement algorithm GRAB.

instance. If any of these counterexamples can be concretized, the property Φ is proved `false`, and the CCE is returned.

If no CCE exists, we start the inner loop over the refinements in this generation. Although the number of ACEs in the SORs decreases monotonically, the length of the SORs does not change in the inner loop. Since all the ACEs have been proven spurious at the very beginning, no concretization test is needed. $\{S_R^L\}$ represents the "reduced SORs." Each time the abstract model is refined, the SORs are reduced (Line 12) until all the spurious counterexamples disappear. Typically, a few that pass through the inner loop produce the breakout, which implies that the set of refinement variables added in the current generation constitutes a sufficient set. A set of refinement variables, when added, kill the entire length-L SORs. Termination is guaranteed by the finiteness of the model.

The game-based heuristic for picking the refinement variables is presented in the next section, followed by SAT-based greedy minimization of the refinement set. Prior art in abstraction-refinement algorithms [5], [7], [8] can be described with a similar framework of pseudo code. However, these algorithms were all based on the analysis of a single ACE. Note that even an optimal refinement algorithm based on a single counterexample cannot necessarily guarantee a good overall refinement. GRAB will be compared to these alternative methods in the experimental-result section.

V. Refinement Variable Selection

We consider the refinement problem as a two-player reachability game in the abstract model. Given the model $\widehat{\mathcal{A}}$ and a

target predicate $\neg p$, the model checking of AGp can be viewed as a two-player concurrent reachability game [44], [45]. The two players are the hostile environment and the abstract model; they play by controlling the values of the pseudoprimary inputs of the abstract model. In $\widehat{\mathcal{A}}$, the pseudoprimary inputs, denoted by $\check{x}$, are the set of invisible variables. Let $\check{x}$ be partitioned into two sets, $\check{x} = w_E \cup w_S$, among which w_E is controlled by the environment (player), and w_S, which is controlled by the system (player).

The positions of the game correspond to the states of the abstract model. Let $\hat{X}$, a valuation of the set of present-state variables $\hat{x}$, be a position; similarly, let capital values of other vector names stand for their valuations. From one position $\hat{X}$, the environment player chooses values for the variables in w_E and, simultaneously, the system player chooses values for variables in w_S. The new position is computed as the unique $\hat{Y}$ satisfying $T(\hat{X}, \check{X}, \hat{Y})$. The goal of the environment player is to go through the spurious paths and reach a state labeled $\neg p$ in spite of the system's opposition. A (memoryless) strategy for the environment player is a function that maps each state of $\widehat{\mathcal{A}}$ to one valuation of the variables in w_E. Likewise, a strategy for the system player is a function that maps each state of $\widehat{\mathcal{A}}$ to one valuation of the variables in w_S.

Definition 1: A position $\hat{X}$ in $\widehat{\mathcal{A}}$ is a winning position for the hostile environment if there exists an environment strategy such that, for all system strategies, $\neg p$ is eventually satisfied.

The concept of winning position is closely related to refinement. Before the abstract model is refined, there are spurious paths from the initial states to states labeled $\neg p$. This corresponds to the partition $(w_E = \check{x}, w_S = \{\ \})$. The hostile environment controls all the invisible variables. Assuming that $\widehat{\mathcal{A}}$ is deterministic, the environment always has a winning strategy because it can force any transition by controlling the variables in $\check{x}$. Refinement can be viewed as removing some variables from w_E and putting them into w_S. Here, we want to identify a small subset of variables that, once removed from w_E, will significantly reduce the number of winning positions for the hostile environment. Here, the two-player reachability game is used to illustrate the intuition behind our refinement algorithm. However, our algorithm can also be viewed as choosing the variable that removes the counterexamples with the largest capacity.

The refinement problem can be stated as follows. Among all the possible partitions of $\check{x} = w_E \cup w_S$, choose the one that gives the environment the least number of winning positions. In this two-player reachability game, the partition that favors the hostile environment the least also favors the abstract system most. Once the partition is available, variables in w_S together with their elementary-transition relations are added into the abstract model.

Given an input variable partition $\check{x} = \{w_E, w_S\}$ and the spurious counterexamples represented by the SORs $\{S^j\}$, the environment's winning positions inside S^j are computed as

$$\exists w_E . \forall w_S . \exists \hat{y} . \left[S^j(\hat{x}) \wedge \widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y}) \right]$$

which is the subset of S^j states from which the environment can force the transition to S^{j+1} despite the opposition from the system.

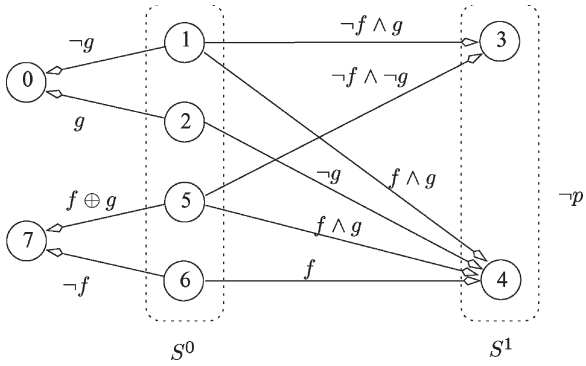


Fig. 9. Illustration of winning positions.

Although universal quantification ($\forall$) is not the same operation as composition, they both reduce the number of enabled edges. Furthermore, it can be shown that when an edge label has an essential variable, a variable which factors out of its label (all the edges in Fig. 9 except the edge from state 5 to state 7), composing that variable with the abstract model splits the abstract edge into two edges (instead of four). Among the two new tail states created by the splits, one has no fan-out nodes—that is, it is a deadend split.

Given the partition $\{w_E, w_S\}$, the normalized number of winning positions for the hostile environment inside S^j are computed as

$$N_j^{\{w_E, w_S\}} = \frac{\left| \exists w_E . \forall w_S . \exists \hat{y} . \left[S^j(\hat{x}) \wedge \widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y}) \right] \right|}{|S^j(\hat{x})|}.$$

Here, $|.|$ stands for the cardinality of the set. N_j, the normalized number of winning positions, is a good indicator of the impact of refining with respect to the variables in w_S. For the purpose of refinement, we prefer the partition that gives N_j the lowest value.

Consider the abstract model in Fig. 9 as an example, in which the first two rings of the SORs are $S^0 = \{1, 2, 5, 6\}$ and $S^1 = \{3, 4\}$, and the set of invisible variables is $\check{x} = \{g, f\}$. When the partition of $\check{x}$ is such that $w_E = \{g\}$ and $w_S = \{f\}$, the set of winning positions for the hostile environment is $\{1, 2\}$. State 1 is a winning position because when the hostile environment makes the assignment $g = 1$, the system player will be forced to a $\neg p$ state (either three or four) no matter what value is assigned to f. A similar argument applies to State 2 as well. According to the definition of N_j

$$N_0^{\{\{g,f\},\{\}\}} = 1.0 \tag{1}$$

$$N_0^{\{\{g\},\{f\}\}} = 0.5 \tag{2}$$

$$N_0^{\{\{f\},\{g\}\}} = 0.25 \tag{3}$$

$$N_0^{\{\{\},\{g,f\}\}} = 0.0. \tag{4}$$

It indicates that g is a better candidate than f for refinement, because putting g alone in w_S gives the hostile environment

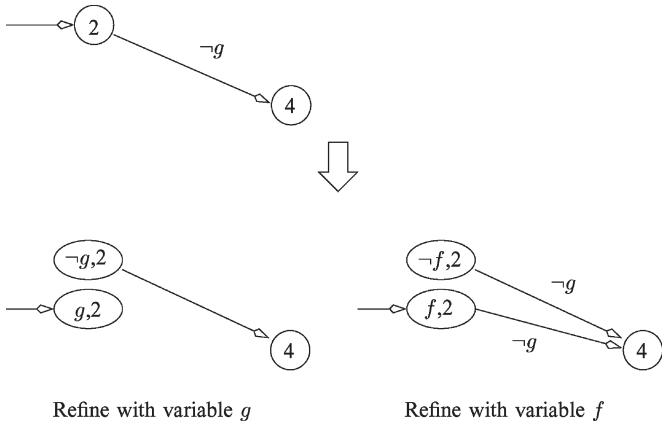


Fig. 10. State splitting: g is better refinement candidate.

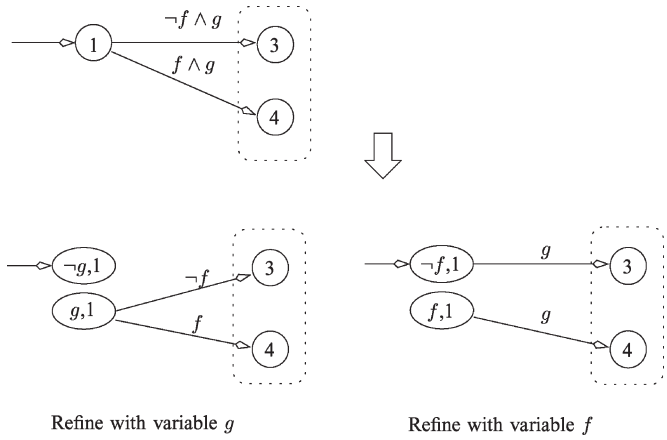


Fig. 11. State splitting: g is still a better refinement candidate.

one winning position, while putting f alone in w_S gives it two winning positions.

To summarize, our refinement algorithm selects a small subset of invisible variables into w_S such that the partition $\{w_E, w_S\}$ minimizes

$$\sum_{0 \le j \le l} N_j^{\{w_E, w_S\}} \qquad \forall \{w_E, w_S\}.$$

This is greedily approximated inside REFINEABSTRACTION. The one variable that minimizes the above number is repeatedly picked (Line 19 in Fig. 8).

The computation of winning positions is similar to BDD-based preimage computation. This computation is scalable because it is performed on the abstract transition relations. By pulling S^j out of the quantifications and using early-quantification [41], [46], the computation can be made very efficient. Furthermore, the following common intermediate result can be shared among different partitions of $\check{x}$:

$$\exists \hat{y} \cdot \left[\widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y}) \right].$$

Note also that $w_E \cup w_S$ contains only invisible variables that are in the local support of the current abstract model, not necessarily the entire set of invisible variables.

A further explanation of the heuristic via state splitting is shown by the two examples in Figs. 10 and 11. In the first example, g is an essential variable to the label on the spurious transition $2 \rightarrow 4$, and f is an irrelevant variable. A variable v is essential to a function $f(v)$ if and only if either $f(0) = 0$ or $f(1) = 0$. By intuition, one would prefer refining with g because f is irrelevant. This is indeed the right choice because it will split State 2 into two new states $(\neg g, 2)$ and $(g, 2)$, only one of which has an out-going edge to State 4. Therefore, it is possible to remove this spurious edge—in the case when State $(\neg g, 2)$ becomes unreachable after refinement. Refining with f, however, does not have such an impact. Both of the two new states $(\neg f, 2)$ and $(f, 2)$ will have out-going edge to State 4. This is consistent with the game-based analysis—State 2 is a winning position for the hostile environment if it controls g.

In the second example, it is no longer easy to figure out that g is still a better refinement candidate than f because both g and f appear in the edge labels (Fig. 11). However, the game-based analysis tells us that State 1 is a winning position for the hostile environment if it controls g. Refining with g produces a similar deadend split—only one of the two new states has an out-going edge to the next ring. Therefore, it is possible to remove this spurious edge, in the case that State $(g, 1)$ becomes unreachable after refinement. Refining with f, however, always leaves the spurious edges intact. This is also consistent with the game-based analysis—State 1 is a winning position for the hostile environment if it controls g.

The refinement method in [31] also relied on analyzing multiple counterexamples. It was based on the classification of invisible variables into strong 0/1 signals and conditional 0/1 signals. A strong 0/1 signal was defined as "in all counterexamples; the value of the signal at the given phase of the trace is zero or one, respectively." In other words, only if a variable is essential with respect to all the label functions of the abstract edges from S^j to S^{j+1} will it be classified as a strong 0/1 signal. In practice, however, this is a very rare case. In fact, both f and g in Fig. 9 are not strong 0/1 signals; according to Glusman *et al.* [31], both would be classified as conditional 0/1 signals and assigned the same weight. Compared to their method, GRAB is often more accurate in identifying important refinement variables. In Fig. 9, for instance, GRAB can tell that g is actually a better refinement candidate than f.

VI. KEEP REFINEMENT SET SMALL

A. Refinement Direction

In our fine-grain abstraction model, there are two types of elementary-transition relations: One is associated with state variables, while the other is associated with BNVs. Accordingly, there are two different refinement directions. If we refine in the sequential direction, we will add more state variables making a potentially larger state space in the refined model; if we refine in the Boolean direction, we will add more logic gates in the fan-in cones of visible state variables, which means that the state space will stay the same but some spurious transitions will be removed. Our experience shows that the

game-based refinement variable selection algorithm itself is not intelligent enough to make the right move. If we do not impose any distinction between the two directions, the final abstract model often contains many redundant state variables. This suggests that refinement needs some guidance on the proper direction.

The idea of predicting the appropriate refinement direction is as follows. If adding BNVs only can remove the spurious counterexamples, then the Boolean direction is chosen to avoid a potentially larger state space; otherwise, the sequential direction is chosen. We predict the proper refinement direction at every refinement step through a satisfiability check similar to the concretization test. The major difference between this check and the multithread concretization test is that it uses the extended abstract model instead of the concrete model. The extended abstract model is defined as the one that contains the visible state variables as well as their complete fan-in logic cones. In Fig. 1, for instance, if the current fine-grain abstract model contains Latch 1 and Gate 5, 7, and 9, then the extended abstract model contains Latch 1, and Gate 1, 2, 4, 5, 7, and 9.

Note that it is the set of BNVs appearing in the extended abstract model but not in the fine-grain abstract model that differentiates the two models. Let $\widehat{T}$ and $\widehat{T}_\epsilon$ be the transition relations of the fine-grain abstract model and the extended abstract model, respectively. If ACEs exist in $\widehat{T}$ but not in $\widehat{T}_\epsilon$, then we should refine in the Boolean direction. Thus, the refinement direction can be decided by solving the Boolean formula $\Psi^\epsilon = \Psi_E \wedge \Psi_S$, where

$$\Psi_E = I_0(X^0) \bigwedge_{0 \le i < L} \widehat{T}_\epsilon(X^i, W^i, X^{i+1})$$

$$\Psi_S = \bigwedge_{0 \le i \le L} S^i(V^i).$$

Ψ_E enables only paths of length L that are allowed by the extended abstract model, while Ψ_S enables only paths in the abstract SORs. Since there exist some length-L counterexamples in the fine-grain abstract model, an unsatisfiable Ψ^ϵ means that it is possible to remove these counterexamples by adding only BNVs in the fan-in cones of the current visible state variables; in this case, we choose the Boolean direction. If Ψ^ϵ is satisfiable, it is impossible to remove all these length-L counterexamples by adding BNVs only. We need to add more state variables by refining in the sequential direction.

B. Refinement Minimization

Once the entire spurious SORs are gone, all the newly added variables form a sufficient refinement set—that is, they are sufficient for removing all the current-length spurious counterexamples. However, this refinement set may not be minimal. Given a sufficient set of refinement variables and the SORs, the refinement-minimization problem can be defined as finding the minimal subset of refinement variables that can kill the spurious counterexamples. In previous work [6], [8], a trial-and-error-based greedy minimization was used to remove the possible redundant variables. This greedy minimization can also be applied here. With fine-grain abstraction, however, minimization must be applied in both refinement directions with respect to the entire SORs instead of a single counterexample.

According to our method for deciding refinement directions, we do not refine in the Boolean direction unless the set of visible state variables becomes sufficient (i.e., no counterexample exists in $\widehat{T}_\epsilon$). As soon as a sufficient set of state variables is added, it is minimized with respect to the entire bundle of counterexamples before refinement shifts to the Boolean direction. When a set of state variables is being minimized, the extended abstraction model induced by these state variables is unrolled to form the SAT formula Ψ_E (referred to the previous section). Every time a state variable is removed from the refinement set, all the BNVs that are relevant only to this state variable are also pruned away. Note that although we have a sufficient set of state variables, ACEs of the current length may still exist in the fine-grain abstract model; although, they do not appear in the extended abstract model any more. After refinement shifts to the Boolean direction, only BNVs will be added until the entire SORs are removed; at this point, the set of newly added BNVs is greedily minimized.

Our refinement minimization resembles the multithread concretization test. The only difference is that, in concretization test, we unroll the concrete model L time frames to capture all length-L concrete paths; in refinement minimization, we unroll the abstract models ($\widehat{T}_\epsilon$ for minimizing the state variables, and $\widehat{T}$ for minimizing BNVs). For every variable in the sufficient set, we first remove it from the set and then check whether the spurious counterexamples come back or not. If they do not come back, that variable is proved to be redundant; otherwise, the variable is necessary and must be added back. Since the satisfiability checks here are conducted in the abstract models, which can be arbitrarily smaller than the concrete model, these SAT problems are usually much easier to solve.

C. Sequential Don't Cares

Previous work in abstraction refinement divided the original system into two parts: a set of visible variables and a set of invisible variables. Model checking was applied to the abstract model that contains only the elementary-transition relations of visible variables. The elementary-transition relations of invisible variables, on the other hand, were completely ignored. Since their transition constraints are removed, the invisible variables are treated as pseudoprimary inputs in model checking, they can take arbitrary values at all times. ACEs may be spurious because the valuations of pseudoprimary inputs that are responsible for triggering these counterexamples may not be allowed in the concrete system.

With some additional analysis of the invisible part of the system, we can further constrain these invisible variables (or pseudoprimary inputs). As is illustrated in Fig. 12, we decompose the invisible part of the system into a series of submodules, each of which contains a subset of the invisible latches. The decomposition is based on the machine-decomposition algorithm originally proposed by Cho *et al.* [23] in the context of reachability analysis. Approximate reachable states of the invisible part can be computed by analyzing each submodule, in turn, assuming that the other submodules are in any

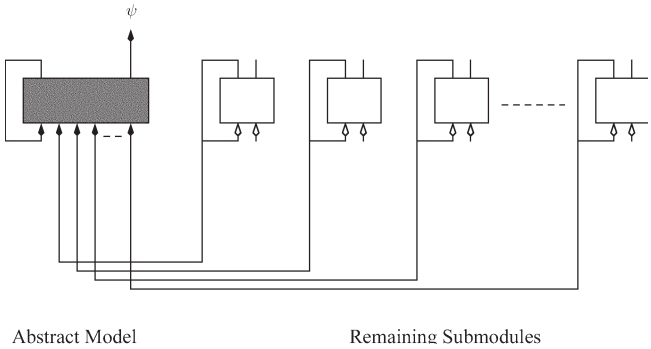


Fig. 12. Sequential don't cares from remaining submodules.

states that have already been estimated to be reachable. The machine-by-machine (MBM) process of reachability analysis is iterated until a least fixpoint is reached. As is pointed out by Moon *et al.* [47], computing approximate reachable states with the MBM algorithm can be several order-of-magnitudes faster than the concrete reachability analysis due to the decoupling of the different submodules.

The set of approximate reachable states of the invisible part computed by MBM is an upper bound on the set of exact reachable states. It can be used to constrain the behavior of the invisible variables of the abstract model. If certain valuations of the invisible variables are not even in the set of approximate reachable states, they will never appear in the original system. Therefore, during the reachability analysis of the abstract model, these pseudoprimary input conditions can be disabled.

In our current implementation, the machine decomposition is applied to the entire system followed by the least fixpoint machine by machine (LMBM) [47] traversal of the submachines, as described above. The approximate reachable states are computed only once before abstraction refinement; they are then used in the abstraction refinement to constrain the forward reachability analysis of the abstract models at every abstraction level. Specifically, the BDD operation constrain [48] is used to remove spurious transitions from $\widehat{T}$ by using the approximate reachable states as the care set. Constraints on the behavior of the abstract model due to the neighboring submachines prevent some spurious ACEs, leading to the decision of a property possibly earlier in the refinement cycle.

Note that a more systematic integration of machine decomposition and approximate reachability analysis into the abstraction-refinement paradigm is also possible. The result would be a multiway partition refinement process. Partitioning of the model into submachines can be done so that the abstract model is one of the many submachines. Refinement is then considered as merging the abstract model with some other submachines.

VII. Experiments

We have implemented our GRAB algorithm and two competing counterexample guided refinement algorithms in VIS-2.0 [35], [36]. We use Colorado University Decision Diagram (CUDD) for the BDD-based computation and Chaff [49] as the back-end SAT solver. The experiments were run under Linux on an IBM IntelliStation with a 1.7-GHz Intel Pentium 4 CPU and 2 GB of RAM. CPU times are in seconds and are all inclusive.

A. Comparison of Refinement Algorithms

Table I compares two variants of the GRAB algorithm against the BDD-based `check_invariant` algorithm in VIS (CI), BMC, the SepSet refinement algorithm [7], a variant of SepSet called SepSet+, and the SAT conflict analysis-based refinement algorithm (CA) of [8]. Here, we focus on comparing the performance of the various refinement variable-selection algorithms; for the purpose of this controlled experiment, the same coarse-grain abstraction and concretization test are used for all abstraction and refinement methods. The CI experiments consist of forward reachability analysis with early termination. For BMC, only the times for failing properties are reported. BMC in VIS checks for one-step inductive invariants, but none of our invariants is one-step inductive. The variant of GRAB, denoted by GRAB, does not perform refinement minimization. The variant SepSet+ differs from SepSet because it minimizes the number of variables in the separation set, instead of the size of the separation tree.

Each model checking run was limited to 8 h. Dynamic variable reordering was enabled (with method sift) for all BDD operations. The comparison was conducted on 14 models, coming from both industry and the VIS verification benchmarks [36].

In Table I, the second column lists the number of binary variables in the cone of influence (COI) of the property. The third column shows the length of the counterexample or of the last ACE encountered by GRAB if the property holds (indicated by a T). For each of the abstraction-refinement methods compared, **iter** is the number of refinement iterations and **regs** is the number of state variables in the proof or disproof. If an experiment ran out of time, the number of iterations performed up to that point and the number of state variables in the last abstract model are given in parentheses. For GRAB, we also report **sat**, which is the time spent in the SAT solver during ACE concretization. Note that in GRAB, **iter** can be larger than **regs** because of refinement minimization.

Note that both variants of the GRAB algorithm significantly outperform CI, SepSet, and CA in terms of CPU time. BMC has the best times for several failing properties, but fails to complete for the hardest problems and for the passing properties. Regarding the size of the BDDs, GRAB is much more efficient than CI; SepSet and CA have even fewer BDD nodes in their model-checking phase, and they use SAT instead of BDDs in the refinement phase.

Table II compares the final abstractions of GRAB and CA. In the table, g is the final set of state variables produced by GRAB, while c is the final set of state variables produced by CA. The first three columns are repeated from Table I.

Table II shows that, in general, there is very good correlation between the final abstractions produced by CA and GRAB. In the 23 experiments that both methods completed, GRAB and CA produced the same final abstraction in four cases. In another ten cases, the abstraction produced by GRAB is strictly better than the one of CA. Conversely, in two cases, CA produces an abstraction that is strictly better than the one of GRAB. These

TABLE I
PERFORMANCE COMPARISON FOR INVARIANT CHECKING ALGORITHMS

circuit	COI	cex	CI	BMC	SepSet			SepSet+			CA			GRAB–			GRAB			
	regs	len	time	time	time	iter	regs	time	iter	regs	time	iter	regs	time	iter	regs	time	iter	regs	sat
D1-p1	101	9	45	**1**	48	11	38	74	9	21	98	15	26	9	18	21	9	18	21	1
D23-p1	85	5	7	**1**	8	2	21	17	2	21	11	1	21	29	5	23	20	5	21	1
D24-p1	147	9	>8 h	27	**1**	0	4	**1**	0	4	**1**	0	4	**1**	0	4	**1**	0	4	1
D24-p2	147	T(9)	>8 h	-	6982	2	8	7087	2	8	2153	34	77	**1**	3	8	3	3	8	1
D1-p2	101	13	1947	**2**	1774	27	45	962	23	38	423	28	44	27	25	28	51	37	23	1
D22-p1	140	10	58	**2**	615	3	133	1005	5	135	728	3	133	537	3	134	720	3	132	1
D1-p3	101	15	1157	**3**	623	22	36	446	19	32	636	25	39	39	23	27	56	34	25	2
D24-p5	147	T(2)	>8 h	-	310	4	7	944	3	7	36	4	11	4	4	6	**3**	4	5	1
D12-p1	48	16	**5**	**5**	106	22	32	124	20	35	64	12	28	6	17	24	14	25	23	1
D2-p1	94	14	166	**6**	147	5	48	280	5	48	239	7	50	124	5	53	180	10	48	1
D16-p1	531	8	837	**10**	>8 h	(35)	(41)	>8 h	(36)	(41)	890	3	16	282	9	14	92	9	14	5
D24-p3	147	T(3)	>8 h	-	>8 h	(1)	(4)	>8 h	(2)	(4)	62	5	11	37	6	8	**20**	6	8	1
D5-p1	319	31	513	58	43	4	13	148	4	13	82	3	13	**26**	9	18	31	9	18	12
D24-p4	147	T(3)	>8 h	-	545	4	7	711	4	7	70	5	11	**29**	6	8	43	6	8	1
D21-p1	92	26	**63**	3787	3790	39	88	2402	36	85	1922	28	79	1010	11	76	2817	26	66	3
B-p1	124	T(18)	7453	-	4359	14	27	4360	14	27	284	5	19	**88**	19	24	173	19	18	6
B-p2	124	17	12988	150	110	2	7	115	2	7	108	2	7	220	8	13	**93**	8	7	11
M0-p1	221	T(3)	>8 h	-	>8 h	(0)	(3)	>8 h	(0)	(3)	1182	9	19	219	14	17	**136**	14	16	20
B-p3	124	T(4)	12466	-	>8 h	(74)	(80)	>8 h	(95)	(101)	167	6	42	**144**	35	52	223	35	43	2
D21-p2	92	28	**152**	10515	4146	36	85	2930	37	86	2962	30	83	2079	19	89	4635	41	70	6
B-p4	124	T(5)	7089	-	9255	49	67	10360	54	68	228	8	43	**157**	36	54	393	47	42	3
B-p0	124	T(17)	7467	-	>8 h	(54)	(61)	>8 h	(39)	(47)	2644	7	49	**330**	28	29	1256	32	24	10
rcu-p1	2453	T(2)	>8 h	-	375	7	11	375	7	11	>8 h	5	(9)	197	9	12	**195**	9	10	0
D4-p2	230	T(19)	765	-	>8 h	(5)	(16)	>8 h	(10)	(22)	>8 h	(3)	(171)	**682**	38	69	1103	69	38	6

TABLE II
CORRELATION BETWEEN FINAL PROOFS (GRAB VERSUS CA)

circuit	COI	cex	$\|g\|$	$\|c\|$	$\|g \cup c\|$	$\|g \cap c\|$	$\|g \setminus c\|$	$\|c \setminus g\|$	subset?
D1-p1	101	9	21	26	27	20	1	6	no
D23-p1	85	5	21	21	21	21	0	0	yes
D24-p1	147	9	4	4	4	4	0	0	yes
D24-p2	147	T(9)	8	77	77	8	0	69	strict
D1-p2	101	13	23	44	44	23	0	21	strict
D22-p1	140	10	132	133	133	132	0	1	strict
D1-p3	101	15	25	39	40	24	1	15	no
D24-p5	147	T(2)	5	11	11	5	0	6	strict
D12-p1	48	16	23	28	28	23	0	5	strict
D2-p1	94	14	48	50	50	48	0	2	strict
D16-p1	531	8	14	16	16	14	0	2	strict
D24-p3	147	T(3)	8	11	13	6	2	5	no
D5-p1	319	31	18	13	18	13	5	0	strict
D24-p4	147	T(3)	8	11	13	6	2	5	no
D21-p1	92	26	66	79	81	64	2	15	no
B-p1	124	T(18)	18	19	19	18	0	1	strict
B-p2	124	17	7	7	7	7	0	0	yes
M0-p1	221	T(3)	16	19	21	14	2	5	no
B-p3	124	T(4)	43	42	43	42	1	0	strict
D21-p2	92	28	70	83	85	68	2	15	no
B-p4	124	T(5)	42	43	43	42	0	1	strict
B-p0	124	T(17)	24	49	49	24	0	25	strict
rcu-p1	2453	T(3)	10	(9)	?	?	?	?	strict
D4-p2	230	T(19)	38	(171)	?	?	?	?	?

differences are in part a consequence of applying refinement minimization once every outer iteration in GRAB, instead of once every inner iteration. The other sources of difference are the order in which variables are selected for refinement (this is what happens in D24-p2) and the order in which they are considered by the greedy-minimization procedure.

Though we exercised diligence in implementing the algorithms of [7] and [8], there remain differences between the originals and our rewritings. For instance, in order to do the controlled experiment, we used the coarse-grain approach when comparing various refinement methods. This is not the case of the original methods of [8] and will, in some cases, impede the search for a good abstraction. However, the drawback is shared by all the methods we implemented and, therefore, should not have a major impact on the comparison we present.

B. Comparing Abstraction Efficiency

Further evidence for the importance of global guidance is provided by an analysis of abstraction efficiency for 80 mid-size test cases from the VIS Benchmarks. Each test case has a passing property and a nontrivial abstract model. It requires at least one refinement iteration. The abstraction efficiency is zero (100%) if the final model contains all (no) state variables. Fig. 13 shows scatter plots of the abstraction efficiency of SepSet, CA, and GRAB. SepSet+ behaves like SepSet. Each

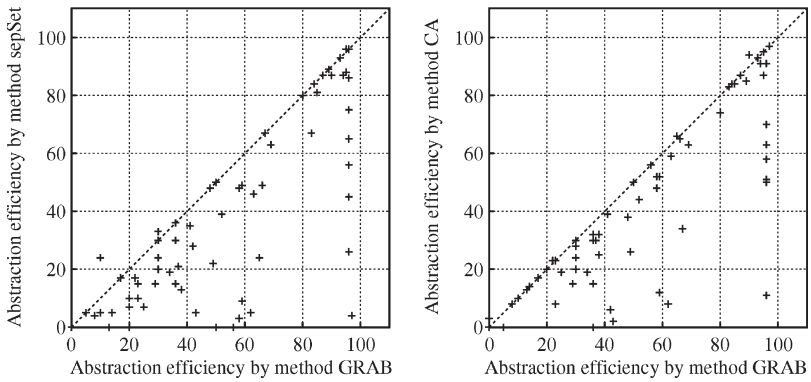


Fig. 13. Comparison of abstraction efficiency. (a) GRAB versus SepSet and (b) GRAB versus CA.

TABLE III
EFFECTIVENESS OF FINE-GRAIN ABSTRACTION AND USE OF SEQUENTIAL DON'T CARES

circuit	COI regs	COI gates	cex len	GRAB time	GRAB regs	+FINEGRAIN time	+FINEGRAIN regs	+ARDC time	+ARDC regs
D1-p1	101	5 k	9	**9**	21	12	20	14	20
D23-p1	85	3 k	5	20	21	**3**	21	14	21
D24-p1	147	8 k	9	1	4	1	4	1	4
D24-p2	147	8 k	T	3	8	3	8	3	8
D1-p2	101	5 k	13	51	23	**27**	23	29	23
D22-p1	140	7 k	10	720	132	242	132	**191**	132
D1-p3	101	5 k	15	56	25	**32**	23	33	23
D24-p5	147	8 k	T	3	5	4	6	**2**	5
D12-p1	48	2 k	16	**14**	23	24	23	19	24
D2-p1	94	18 k	14	180	48	108	49	**59**	48
D16-p1	531	34 k	8	92	14	25	14	**21**	14
D24-p3	147	2 k	T	20	8	4	6	**2**	5
D5-p1	319	25 k	31	**31**	18	42	13	32	13
D24-p4	147	8 k	T	43	8	4	6	**2**	5
D21-p1	92	14 k	26	2817	66	2725	70	**622**	67
B-p1	124	2 k	T	173	18	189	19	**159**	18
B-p2	124	2 k	17	93	7	95	7	**90**	7
M0-p1	221	29 k	T	**136**	16	204	13	942	13
B-p3	124	2 k	T	223	43	76	43	**62**	43
D21-p2	92	14 k	28	4635	70	1748	75	**868**	67
B-p4	124	2 k	T	393	42	**101**	43	108	42
B-p0	124	2 k	T	**1256**	24	1507	24	1484	24
rcu-p1	2453	38 k	T	195	10	**188**	10	216	10
D4-p2	230	8 k	T	1103	38	204	38	**195**	38
IU-p1	4494	154 k	T	>8 h	-	**2226**	12	2263	12
IU-p2	4494	154 k	T	>8 h	-	930	14	**699**	12
Total				>16 h +12207		10724		8130	

point below the diagonal represents a win for GRAB. Scatter plots for the other pairs of methods (not shown for lack of space) show no clear winner.

Refinement minimization, though essential for good performance of CA, does not always improve CPU time when applied to our refinement scheme. The time spent checking the variables for redundancy and the additional iterations are not always offset by the reduction in the size of the abstraction. Nonetheless, we argue that as we progress toward larger models, refinement minimization adds to the robustness of the method.

C. Fine-Grain Abstraction and Sequential Don't Cares

Experiments were also conducted to test the effectiveness of fine-grain abstraction and the use of sequential don't cares. In the implementation of the fine-grain abstraction, we set the BDD threshold of frontier partitioning to 1000. Every time the BDD size of the transition function went beyond this threshold, a BNV was inserted in the combinational logic cones. We have found that setting the BDD threshold to 1000 is consistently better when dealing larger models; although, the default threshold 5000 works well for smaller ones (such as VIS's own test suite).

The first four columns of Table III repeat the statistics of the test cases. The first column shows the names of the designs and the second and third columns give the numbers of binary state variables and logic gates in the COI, respectively. The fourth column indicates whether the properties are `true` (T) or `false` (F). If the properties are `false`, the lengths of the shortest counterexamples are given. The following six columns

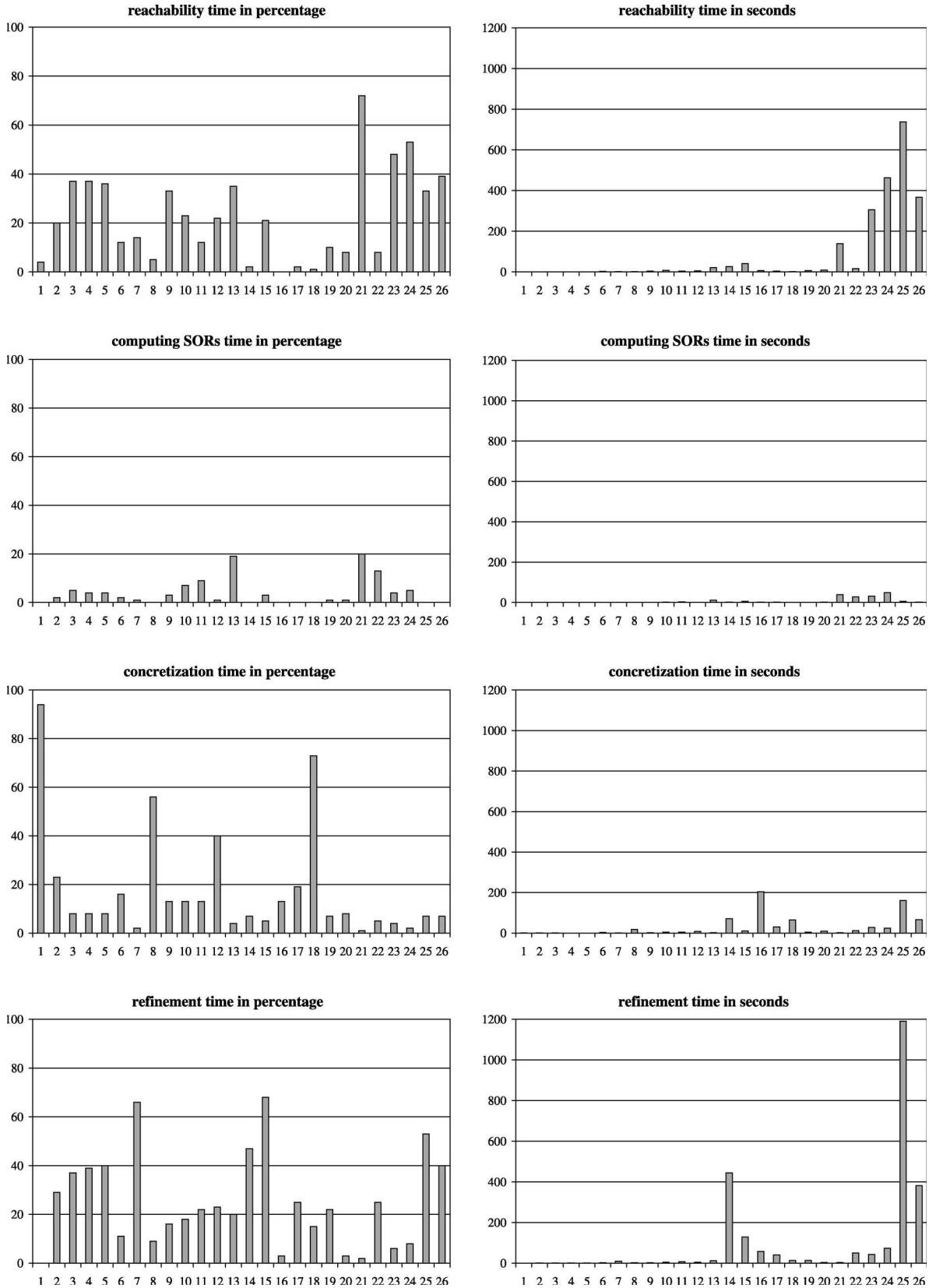


Fig. 14. CPU time spent on forward reachability analysis, computing SORs, multithread concretization test, and refinement with GRAB.

compare the performance of three different implementations: GRAB uses the coarse-grain abstraction, +FINEGRAIN is GRAB plus the fine-grain abstraction method, and +ARDC is GRAB plus fine-grain abstraction and sequential Don't Cares. The underlying algorithm for picking refinement variables is the same for the three methods. For each method, the

CPU time in seconds and the number of state variables in the final abstract model are shown.

The fine-grain-abstraction approach shows a significant performance improvement over the conventional coarse-grain approach. First, it is able to finish the two largest test cases that cannot be verified otherwise. Careful analysis of IU-p1 and IU-p1, two problems from the instruction unit of the PicoJava microprocessor, shows that some of their registers have extremely large fan-in combinational logic cones. Without fine-grain abstraction, abstract models with less than ten registers would have been too complex for the model checker to deal with. For the other test cases that both methods managed to finish, +FINEGRAIN is significantly faster than GRAB. In fact, the total CPU time required to finish the 24 remaining test cases is 12 207 s for GRAB, and 7562 s for +FINEGRAIN.

With the use of sequential don't cares, the performance of +FINEGRAIN is further improved. +ARDC is significantly faster than both +FINEGRAIN and GRAB on more than half of the 26 test cases and is also comparable for the remaining ones. The total CPU time required to finish all the 26 test cases is 10 724 s for +FINEGRAIN and 8130 s for +ARDC; this is an average of 25% speedup.

D. Runtime Breakdown in Abstraction Refinement

Fig. 14 shows the allocation of CPU time among the different phases in abstraction refinement. These data were extracted with +ARDC; therefore, they correspond to the last column in Table III. The four figures at the left-hand side give in percentage the CPU time spent on reachability analysis, on computing the SORs, on the multithread concretization test, and on computing the refinement with GRAB, respectively. The four figures at the right-hand side give the corresponding CPU time in seconds. The 26 test cases are listed on the x-axis in all figures. Note that other things also consume part of the CPU time, such as incrementally building the BDD partitions, the creation, and deletion of abstract finite-state machine (FSM), etc.

Fig. 14 demonstrates that forward reachability analysis and computing the refinement with GRAB have consumed most of the CPU time. The backward reachability analysis to build the SORs, on the other hand, often takes significantly less time than its forward counterpart, even though it collects all the shortest counterexamples. This is due to the application of forward onion rings as care sets in the corresponding preimage computations. Furthermore, the actual run time of the concretization test is often small (as shown by the "in seconds" figure), even though it takes a significant amount in percentage from the total CPU time (as shown by the "in percentage" figure). On this particular set of test cases, multithread concretization test is never the performance bottleneck. On the harder problems, test cases 19–26, its overhead becomes negligible.

The performance of forward reachability analysis is limited by the capacity of the state-of-the-art BDD-based symbolic techniques. As the abstract model gets larger, BDD-based computations become more and more expensive. The size the abstract model also affects the overhead of the GRAB refinement algorithm. The size of the BDDs for representing the SORs becomes larger as the model gets more complex. In addition, a larger abstract model often has more invisible variables in its local support, which means that more CPU time needs to be spent on scoring them.

VIII. Conclusion

Recent abstraction-refinement research and the advances in BDDs and SAT solvers have led to model-checking algorithms that exhibit much increased robustness on problems with hundreds of state variables and are beginning to foray into the thousands of variables. The combination of decision procedures that characterize those methods raises the issue of global versus local guidance in the search for counterexamples. In this paper, we have shown that significant performance improvements can be achieved by emphasizing global guidance, smaller abstraction granularity, and the use of don't cares of invisible variables.

Our experience with real-world designs shows that many of them have a large number of state variables and very complex combinational logic cones. To build meaningful yet still tractable abstract models for these complex systems, fine-grain abstraction has been proved to be indispensable. Furthermore, emphasizing global guidance in refinement pays off. Our SOR-based approach, based on the analysis of all shortest counterexamples, often achieves a much higher abstraction efficiency relative to methods based on one counterexample only. Since our GRAB refinement variable-selection algorithm is executed on the abstract model and its local support variables only, it is also more scalable than many of the previous methods that require computation on the concrete system. We have found that the cost of concretization test for multiple counterexamples is usually less than the cost of SAT-based refinement procedures. In addition, a practical lessening of the concretization check problem may also come from an incremental approach like the one in [9].

References

[1] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi, "Improving Ariadne's bundle by following multiple threads in abstraction refinement," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2003, pp. 408–415.

[2] C. Wang, G. D. Hachtel, and F. Somenzi, "Fine-grain abstraction and sequential don't cares for large scale model checking," in *Proc. Int. Conf. Comput. Des.*, San Jose, CA, Oct. 2004, pp. 112–118.

[3] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop Logics Programs*. Berlin, Germany: Springer-Verlag, 1981, vol. 131, *LNCS*, pp. 52–71.

[4] K. L. McMillan, *Symbolic Model Checking*. Boston, MA: Kluwer, 1994.

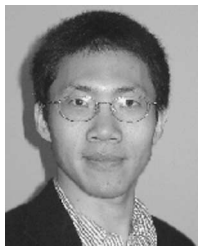
[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. CAV*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Germany: Springer-Verlag, Jul. 2000, vol. 1855, *LNCS*, pp. 154–169.

[6] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano, "Formal property verification by abstraction refinement with formal, simulation and hybrid engines," in *Proc. Des. Autom. Conf.*, Las Vegas, NV, Jun. 2001, pp. 35–40.

[7] E. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning," in *Proc. CAV*, E. Brinksma and K. G. Larsen, Eds. Berlin, Germany: Springer-Verlag, Jul. 2002, vol. 2404, *LNCS*, pp. 265–279.

[8] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Formal Methods in Computer Aided*

Design, vol. 2517, *LNCS*, M. D. Aagaard and J. W. O'Leary, Eds. New York: Springer-Verlag, Nov. 2002, pp. 33–51.
[9] S. Barner, D. Geist, and A. Gringauze, "Symbolic localization reduction with reconstruction layering and backtracking," in *Proc. CAV*, E. Brinksma and K. G. Larsen, Eds. Berlin, Germany: Springer-Verlag, Jul. 2002, vol. 2404, *LNCS*, pp. 65–77.
[10] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Proc. TACAS*, Warsaw, Poland, Apr. 2003, vol. 2619, *LNCS*, pp. 2–17.
[11] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Learning from BDDs in SAT-based bounded model checking," in *Proc. Design Automation Conf.*, Jun. 2003, pp. 824–829.
[12] B. Li, C. Wang, and F. Somenzi, "A satisfiability-based approach to abstraction refinement in model checking," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 143–155, 2003. International Workshop on Bounded Model Checking. [Online]. Available: http://www.elsevier.nl/locate/entcs/volume89.html
[13] B. Li and F. Somenzi, "Efficient computation of small abstraction refinements," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2004, pp. 518–525.
[14] B. Li, C. Wang, and F. Somenzi, "Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure," *Softw. Tools Technol. Transf.*, vol. 2, no. 7, pp. 143–155, 2005.
[15] L. Zhang, M. R. Prasad, and M. S. Hsiao, "Incremental deductive and inductive reasoning for SAT-based bounded model checking," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 2004, pp. 502–509.
[16] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle, "Dynamic abstraction using SAT-based BMC," in *Proc. ACM/IEEE Des. Autom. Conf.*, San Jose, CA, Jun. 2005, pp. 754–757.
[17] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints," in *Proc. ACM Symp. Princ. Program. Lang.*, 1977, pp. 238–250.
[18] R. Milner, "An algebraic definition of simulation between programs," in *Proc. 2nd Int. Joint Conf. Artif. Intell.*, 1971, pp. 481–489.
[19] D. L. Dill, A. J. Hu, and H. Wong-Toi, "Checking for language inclusion using simulation relations," in *Proc. CAV*, K. G. Larsen and A. Skou, Eds. Berlin, Germany: Springer-Verlag, Jul. 1991, vol. 575, *LNCS*, pp. 255–265.
[20] K. Fisler and M. Y. Vardi, "Bisimulation and model checking," in *Proc. CHARME*. Berlin, Germany: Springer-Verlag, Sep. 1999, vol. 1703, *LNCS*, pp. 338–341.
[21] F. Balarin and A. L. Sangiovanni-Vincentelli, "An iterative approach to language containment," in *Proc. CAV*, C. Courcoubetis, Ed. Berlin, Germany: Springer-Verlag, 1993, vol. 697, *LNCS*, pp. 29–40.
[22] D. E. Long, "Model checking, abstraction, and compositional verification," Ph.D. dissertation, Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, Jul. 1993.
[23] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for approximate FSM traversal based on state space decomposition," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 15, no. 12, pp. 1465–1478, Dec. 1996.
[24] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton Univ. Press, 1994.
[25] R. H. Hardin, Z. Har'El, and R. P. Kurshan, "COSPAN," in *Proc. CAV*, T. Henzinger and R. Alur, Eds. Berlin, Germany: Springer-Verlag, 1996, vol. 1102, *LNCS*, pp. 423–427.
[26] A. Pnueli, "The temporal logic of programs," in *Proc. IEEE Symp. Found. Comput. Sci.*, Providence, RI, 1977, pp. 46–57.
[27] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, "Tearing based abstraction for CTL model checking," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 1996, pp. 76–81.
[28] J. Lind-Nielsen and H. R. Andersen, "Stepwise CTL model checking of state/event systems," in *Proc. CAV*, N. Halbwachs and D. Peled, Eds. Berlin, Germany: Springer-Verlag, 1999, vol. 1633, *LNCS*, pp. 316–327.
[29] A. Pardo and G. D. Hachtel, "Incremental CTL model checking using BDD subsetting," in *Proc. Des. Autom. Conf.*, San Francisco, CA, Jun. 1998, pp. 457–462.
[30] J.-Y. Jang, I.-H. Moon, and G. D. Hachtel, "Iterative abstraction-based CTL model checking," in *Proc. DATE*, Paris, France, Mar. 2000, pp. 502–507.
[31] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi, "Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation," in *Proc. TACAS*, Warsaw, Poland, Apr. 2003, vol. 2619, *LNCS*, pp. 176–191.
[32] F. Y. C. Mang and P.-H. Ho, "Abstraction refinement by controllability and cooperativeness analysis," in *Proc. Design Automation Conf.*, San Diego, CA, Jun. 2004, pp. 224–229.
[33] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[34] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proc. DATE*, Munich, Germany, Mar. 2003, pp. 880–885.
[35] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Proc. CAV*, T. Henzinger and R. Alur, Eds. Berlin, Germany: Springer-Verlag, 1996, vol. 1102, *LNCS*, pp. 428–432.
[36] [Online]. Available: http://vlsi.colorado.edu/~vis
[37] A. Gupta, M. K. Ganai, and P. Ashar, "Lazy constraints and SAT heuristics for proof-based abstraction," in *Proc. Int. Conf. VLSI Des.*, Jan. 2005, pp. 183–188.
[38] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of sat-based model checking techniques in an industrial environment," in *Proc. CHARME*, Oct. 2005, pp. 254–268.
[39] N. Amla and K. L. McMillan, "A hybrid of counterexample-based and proof-based abstraction," in *Proc. Formal Methods Comput. Aided Design*, Nov. 2004, pp. 260–274.
[40] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.
[41] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley, "Efficient BDD algorithms for FSM synthesis and verification," presented at Int. Workshop Logic Synthesis (IWLS), Lake Tahoe, CA, May 1995.
[42] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs complement SAT-based BMC in DiVer," in *Proc. CAV*, W. A. Hunt, Jr. and F. Somenzi, Eds. Berlin, Germany: Springer-Verlag, Jul. 2003, vol. 2725, *LNCS*, pp. 206–209.
[43] G. Cabodi, S. Nocco, and S. Quer, "Improving SAT-based bounded model checking by means of BDD-based approximate traversal," in *Proc. Conf. Des. Autom. Test Eur.*, Munich, Germany, Mar. 2003, pp. 898–905.
[44] E. A. Emerson and C. S. Jutla, "Tree automata, mu-calculus and determinacy," in *Proc. 32nd IEEE Symp. Found. Comput. Sci.*, Oct. 1991, pp. 368–377.
[45] H. Jin, K. Ravi, and F. Somenzi, "Fate and free will in error traces," in *Proc. TACAS*, Grenoble, France, Apr. 2002, vol. 2280, *LNCS*, pp. 445–459.
[46] I.-H. Moon, J. H. Kukula, K. Ravi, and F. Somenzi, "To split or to conjoin: The question in image computation," in *Proc. Design Automation Conf.*, Los Angeles, CA, Jun. 2000, pp. 23–28.
[47] I.-H. Moon, J. Kukula, T. Shiple, and F. Somenzi, "Least fixpoint approximations for reachability analysis," in *Proc. Int. Conf. Comput.-Aided Des.*, San Jose, CA, Nov. 1999, pp. 41–44.
[48] O. Coudert, C. Berthet, and J. C. Madre, "Formal Boolean manipulations for the verification of sequential machines," in *Proc. Eur. Conf. Design Automation*, Mar. 1990, pp. 57–61.
[49] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. Design Automation Conf.*, Las Vegas, NV, Jun. 2001, pp. 530–535.



Chao Wang received the B.S. degree from Peking University, Beijing, China, in 1996, and the Ph.D. degree from the University of Colorado, Boulder, in 2004, both in electrical engineering.

He is a Research Staff Member of NEC Laboratories America, Princeton, NJ. His research includes formal specification and verification of concurrent systems (hardware, software, and embedded systems).

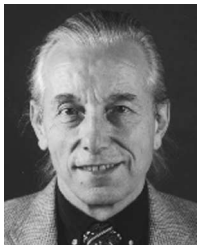
Dr. Wang is a recipient of the 2003–2004 Association of Computing Machinery (ACM) Outstanding Ph.D. Dissertation Award in Electronic Design Automation.

Bing Li received the B.S. and M.S. degrees from Huazhong University of Science and Technology, Wuhan, China, in 1996 and 1999, respectively, and the M.S. degree from the University of Colorado, Boulder, in 2001, all in electrical engineering. He is currently working toward the Ph.D. degree in the ECE Department of University of Colorado.

His research includes formal verification, with a focus on algorithms based on binary decision diagrams (BDD) and satisfiability (SAT).

HoonSang Jin received the Ph.D. degree in electrical engineering from the University of Colorado, Boulder, in 2005.

He is currently with Samsung Electronics, Korea. His research interests include electronic design automation, with a focus on formal verification using BDDs, Boolean satisfiability solvers, and circuit-based reasoning.



Gary D. Hachtel (S'62–M'65–SM'74–F'80–LF'04) received the B.S. degree from California Institute of Technology, Pasadena, in 1959, and the Ph.D. degree from the University of California, Berkeley, in 1964, both in electrical engineering.

He is a Professor Emeritus who has been working for the last 20 years in the University of Colorado, Boulder, in fields of logic synthesis and formal verification. For the 17 years prior to his stint at the University of Colorado, he was a Research Staff Member in the Department of Math Sciences, IBM Research, Yorktown Heights, NY.

Dr. Hachtel received the IEEE CASS Mac Van Valkenburg Award in 2004, for his distinguished career of fundamental innovations across the broad spectrum of semiconductor electronic design automation.

Fabio Somenzi received the Dr. Eng. degree in electronic engineering from Politecnico di Torino, Turin, Italy, in 1980.

He is a Professor in the ECE Department, University of Colorado, Boulder. He has published one book and over 140 papers on the synthesis, optimization, verification, simulation, and testing of digital systems. Prior to joining the University of Colorado in 1989, he was with SGS-Thomson Microelectronics, Italy, managing a team for computer aids for digital design.

Dr. Somenzi has served as Associate Editor for IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN and the *Springer Journal on Formal Methods in Systems Design*, and on the program committees of premier EDA conferences including International Conference on Computer Aided Design (ICCAD), digital-to-analog converter (DAC), International Conference on Computer Design (ICCD), European Design Automation Conference (EDAC)/Design, Automation and Test in Europe (DATE), International Workshop for Logic Synthesis (IWLS), and International Symposium on Low Power Electronics and Design (ISLPED). He was the Conference Cochair of Computer Aided Verification (CAV) in 2003.