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

Chao Wang Bing Li HoonSang Jin Gary D. Hachtel Fabio Somenzi

Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO 80309-0425

{wangc,bli,jinh,hachtel,fabio}@Colorado.EDU

Abstract

We propose an abstraction refinement method for invariant checking, which is based on the simultaneous analysis of all abstract counter examples of shortest length in the current abstraction. The algorithm is focused on an *improved Ariadne's Bundle* [1] of SORs (*Synchronous Onion Rings*) of the abstract model; the transitions through these SORs contain all shortest ACEs (Abstract Counter Examples) and no other ACEs. The SORs are exploited in two distinct ways to provide global guidance to the abstraction refinement process: (1) Refinement variable selection is based on the entirety of transitions connecting the SORs, and (2) a SAT-based concretization test is formulated to test all ACEs in the SORs at once. We call this test multi-thread concretization. The scalability of our refinement algorithm is ensured in the sense that all the analysis and computation required in our refinement algorithm are conducted on the abstract model.

The *abstraction efficiency* of a given abstraction refinement algorithm measures how much of the concrete model is required to make the decision. We include experimental comparisons of our new method with recently published techniques [6, 4]. The results show that our scalable method, based on global guidance from the entire bundle of shortest ACEs, outperforms these other methods in terms of both run time and abstraction efficiency.

1. Introduction

The primary obstacle to widespread use of formal verification techniques, especially contemporary symbolic model checking programs, 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 explosive growth is increasing use of high level HDLs (Hardware Description Languages); 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, including this one, have shown that symbolic model checkers, extended to include an automated abstraction refinement paradigm, still hold great promise in dealing with state explosion.

In such a paradigm, one seeks the simplest abstract model sufficient to decide the property at hand. This corresponds to achieving the maximium abstraction efficiency. However, the optimum abstraction problem is hard [5], and existing practical abstraction and refinement techniques do not guarantee or even pursue global optimality.

Automated abstraction technique was first introduced by Kurshan [16] in checking linear properties specified by ω-regular automata, where overapproximation suffice to prove a property true. In COSPAN [11], the initial abstraction contains only the state variables in the property and leaves the others unconstrained; the counter example obtained from the abstract model was searched for the variables appearing in it and mincut-maxflow eventually determined which variables of the model comprised the refinement step.

For logics like CTL [7] or the full μ-calculus [15], which express also existential and mixed properties, one has to resort to both underapproximation and overapproximation [14, 20, 12]. If one wants to prove a universal property false, one can try to concretize the counter example as done in [16, 5], or use underapproximation as in [17]. Concretization via satisfiability was introduced in [22] (where an ATPG program was used) and improved in [6, 4].

However, in these existing methods, only a single abstract counter example is used for concretization test and for deriving the refinement; this single ACE is usually arbitrarily chosen. One can argue that a drawback of these methods is their "needle in a haystack" approach, lacking global criteria or search strategies; we have observed, on a real-world circuit, that the number of the shortest ACEs can be larger than 10^{45}. Focusing on a single ACE among such an astronomically large number of ACEs makes it difficult to find a set of refinement variables sufficient to kill all ACEs of the current length L and thus might lead the refinement in the wrong direction; it might also be difficult to find a real counter example by the concretization test.

In contrast, we propose a new refinement algorithm that works on global guidance provided by the "improved Ariadne's Bundle" of all shortest ACEs. This bundle is represented by the set of SORs (Synchronous Onion Rings) of the abstract model. Each onion ring is the intersection of forward and backward reachable onion rings at the same number of time step from (or to) the initial states. In case all of the shortest ACEs are spurious, the refinement process

*This work was supported in part by SRC contract 2002-TJ-920 and NSF grant CCR-99-71195.

[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."

.

tries to find the minimal set of refinement variables sufficient to kill all the given ACEs (that is, sufficient to prove that all ACEs of the current set of SORs are spurious).

The algorithm does not try to kill the entire bundle in one shot; rather, it tries to identify local variables that are critical to the refinement according to a game theoretic approach to exploiting the global guidance provided by the SORs. It may take a set of refinement steps, called a *generation* of refinements, before all the shortest ACEs at a given length disappear. From our experience, typically only a few such iterations are required.

Like [4], the new algorithm also employs a form of the refinement minimization first introduced in [22]. However we make this task more effective by a technique called *generational refinement minimization*. This method performs minimization based on testing the non-emptiness of the SORs, and is triggered only at the end of each refinement generation. Note that at the point we have a set of refinement variables, added in this generation, that are sufficient to kill the entire ACEs in the SORs. We use a greedy approach that first tries to drop some of these variables and then checks if the SORs can still be killed, thus minimizing the set of refinement variables in this generation.

Our SAT-based multi-thread concretization test decides with one satisfiability check whether *any* ACE in the SORs can be concretized. Data show that the time to conduct this test is surprisingly close to the concretization test time for a single ACE, and usually much less than the overall verification time.

In summary, our method differs from previous counter example driven abstraction refinement schemes [5, 22, 6, 4] in that: (1) It handles all shortest ACEs, rather than a single shortest ACE; (2) at each ACE level in the concretization test, a set of abstract states, instead of just one abstract state, are used to constrain the bounded unrolled concrete model at each time step; and (3) the refinement is based on the analysis of all the spurious ACEs with a game theoretic approach. Since this refinement variable selection method operates solely on the abstract model and its local support variables, it is more scalable than those methods that involve computation on the concrete model.

More recently, an automatic abstraction algorithm based on the unsatisfiable BMC runs was proposed [18], and it also can eliminate all the counter examples. However, our refinement algorithm is BDD-based and is conducted solely on the abstract models, whereas theirs is based on computing the unsatisfiable proof for concrete BMC instances using SAT. Furthermore, our multi-thread concretization test is also different from their plain BMC runs, for we use the SORs to restrict the search space. In another recent paper [9], *multi-valued counter example* was used to guide the refinement. This multi-valued annotation collects information from multiple counter examples, but is still different from our SORs in the sense that it does not capture all the shortest ACEs. (For example, their concretization test does not check all the shortest ACEs.)

We present a thorough experimental comparison of our new algorithm to the BDD-based invariant check algorithm in VIS [3], VIS's BMC (Bounded Model Checking), the SepSet algorithm of [6] and the Conflict Analysis (CA) approach of [4]. For the purpose of experimental comparision, we also implemented the algorithms of [6, 4]. The experiments were conducted on circuits from both public-domain and industry. Many of the models used were kindly provided by the authors of [4]. After establishing notation, we discuss how to compute and deploy Ariadne's bundle in the abstraction refinement process. This leads to a discussion of the new algorithm, its comparison to our implementation of competing methods, and indications for promising new avenues of research.

2. Preliminaries

In this section we establish the basic properties of the considered abstractions, and then define and illustrate the SORs.

A model is given in terms of: (1) A set of present-state variables $x = \{x_1,...,x_m\}$; (2) a set of input variables $w = \{w_1,...,w_n\}$; and (3) a set of next-state variables $y = \{y_1,...,y_m\}$. Thus, the model M can be represented by the pair $\langle T, I\rangle$, where $T(x,w,y)$ is the transition relation, and $I(x)$ is the set of initial states. We assume that the concrete model is the synchronous composition of a collection of m elementary components, each of which contains exactly one state variable. Let $J = \{1,\ldots,m\}$; then,

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

where $T_j(x,w,y_j)$ is the transition relation of the jth binary state variable, and thus depends on one next-state variable. $T_j(x,w,y_j)$ is defined as $[y_j \leftrightarrow \delta_j(x,w)]$, where $\delta_j(x,w)$ is the transition function of the jth state variable.

The abstract model consists of $k \le m$ elementary components. Let $\hat{J} = \{1,\ldots,k\} \subseteq J$. Then,

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

Here $\hat{y} = \{y_j \mid j \in \hat{J}\}$ is the subset of next-state variables in the abstract model. Let $\hat{x}$ denote the subset of present-state variables corresponding to $\hat{y}$, x can be bipartitioned into $\hat{x}$ (state variables in the abstract model) and $\check{x} = x \setminus \hat{x}$ (state variables that are abstracted away). During the symbolic model checking of the abstract model, variables in $\check{x}$ are treated as inputs; $\hat{w} = \{\check{x}, w\}$ is now the entire set of input variables for the abstract model. $\hat{M} = \langle \hat{T}(\hat{x},\hat{w},\hat{y}), \hat{I}(\hat{x})\rangle$ represents an over-approximation of the original model. Informally, $\hat{I}$ is an existential projection of I: An abstract state is called initial if it contains a concrete initial state.

In the sequel, we will assume that the given property is an invariant $\mathsf{AG}\, p$ (that is, "p holds globally on all paths"). The states satisfying the propositional property p are also abstracted by projection: If an abstract state contains a concrete state that is labeled by p, then it too will be labeled by p. Thus the abstract model $\hat{M}$ simulates the concrete model M, and, if the property passes on the abstract model, it also passes on the concrete model. However, if the property fails and an abstract counter example is generated from an initial state to a $\neg p$ state, it may or may not be a CCE (concrete counter example). A concretization test determines whether an abstract counter example is real or spurious. In the published works [6, 4], given an ACE $s_0 s_1 \ldots s_L$, such a concretization test builds a length-L bounded concrete model (unrolling the model exactly L time steps) and constrains it at each time step with the abstract state s_i; a satisfiability check is then conducted on this constrained bounded model, either by SAT-solvers or by ATPG engines. If the ACE is concretizable (i.e., there exists a satisfiable assignment), the property fails with a real counter example; otherwise, the result is still inconclusive and we have to refine the abstract model, by composing more elementary transition relations.

Our new algorithm is based on the synchronous onion rings discussed below, which are obtained by forward and backward traversing the state transition graph of the model.

The abstraction we call "Ariadne's Bundle" is a subset of the transitions of $\hat{T}$, as illustrated in Fig. 1. The state transition graph of the abstract model $\hat{M}$ is shown in Part (1). Abstract forward reachability from $\hat{I}$ gives the indicated sets of states called the forward onion rings: $F^1 = \{3,4,5\}$, etc.. Note that $\neg p$ is first reached at the 3rd step of the search. An analogous backward reachability

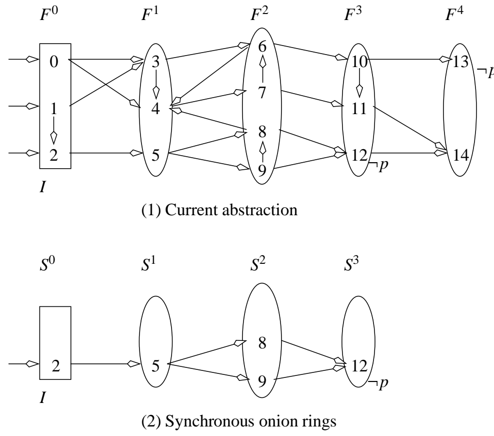

Figure 1: Ariadne's bundle of synchronous onion rings.

from the $\neg p$ state in the 3rd step would reach states $\{8,9\}$ at the 2nd step, $\{5\}$ at the 1st step, and the initial state 2. Both forward and backward abstract onion rings induce subrelations of $\hat{T}$, and thus can be used to restrict the search space on the concrete model [10]. Taking the intersection of the two sets of states at each step gives the synchronous onion rings

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

The term "Ariadne's bundle" refers to the subrelation T^B induced by considering only the transitions of $\hat{T}$ between a state at one step to another state in the next step in the SORs. It comprises the bundle of all shortest ACEs, and no other ACEs. In this simple case there are two shortest ACEs, both of length 3. In practice, however, the number of ACEs in the SORs is typically astronomically large.

Comparing the state transition graph of Ariadne's bundle to the original state transition graph, one can note substantial reduction in the number of states and transitions. Thus while the abstraction $\hat{T}$ can be much simpler than the concrete system T, Ariadne's bundle can be much simpler still.

3. Generational Refinement

We begin illustrating the generic process of the abstraction and refinement, by treating the simple example in Fig. 2. The concrete model M is the synchronous composition of three components: M_1, M_2, and M_3. That is,

$$M = M_1 \parallel M_2 \parallel M_3\ \ .$$

Each component M_i has one state variable v_i. In M_1, the state variable v_1 can take 4 values and thus must be implemented by 2 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 $AG(v_1 \neq 3)$, i.e., State 3 in M_1 is not reachable. The right part in Fig. 2 gives the state transition graph of the concrete model M. It is not difficult to see that the property fails on the concrete model, as shown by the bold edges, which exhibits the length-4 CCE (Concrete Counter Example): (000, 111, 200, 211, 300).

Let the initial abstraction be $\hat{M} = M_1$, that is, only the state variable appearing in the given property is preserved, and all the other state variables are treated as inputs. Note that there is an abstract counter-example of length 3: (0_ _, 1_ _, 2_ _, 3_ _); this ACE is spurious because it is not concretizable on M.

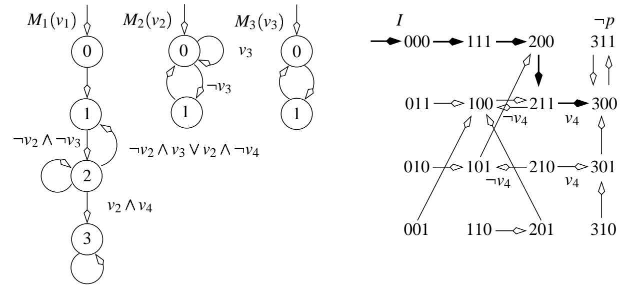

Figure 2: A simple example.

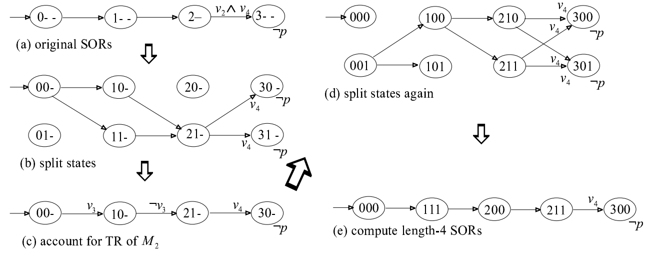

Figure 3: The refinement process.

Although this example is extremely simple, it demonstrates an important aspect of the abstraction/refinement process. Refinement mechanisms like those in [5, 6], since they are actuated by a single ACE, might pick only variable v_2 for refinement. However, after this refinement, an ACE of length 3 still exists; for example, it could be (00_, 10_, 21_, 30_). This shows that $\{v_2\}$ is not a sufficient refinement set to kill the ACE (0_ _, 1_ _, 2_ _, 3_ _). This is suggestive of the danger of placing too much refinement emphasis on a single ACE. Of course, this is much more of a problem when the SORs contain an extremely large number of shortest ACEs. In this case choosing the refinement variables on the basis of a single ACE could be ineffective.

We now discuss and illustrate the proposed framework of generational SOR-based refinement, using the above example. Note that in building the SORs, self loops and back edges are pruned away to focus on the shortest ACEs in the current abstract model. In the initial abstract model $\hat{M} = M_1$, all the shortest ACEs are of length 3, and the SORs are just the 4 states of M_1, connected by the 4 forward edges.

Because the first *generation* of shortest ACEs are of length 3, we start our refinement process by dealing with all these length-3 SORs until all of them are killed. These initial SORs are shown in Part (a) of Fig. 3. Note that in M_1 only edges from state 2 to states 1, 2, and 3 are labeled. As discussed below in Section 5, these labels cause our variable selection routine to pick variable v_2 for the first refinement. The refined abstract model will be $\hat{M} = M_1 \parallel M_2$; however, $\hat{M}$ 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 Part (b) of Fig. 3. Because of the label $v_2 \wedge v_4$, the last abstract edge, (2_ _,3_ _), is split into 2, rather than 4 refined edges. State 20_ is now backward unreachable from $\neg p$, so its two incoming edges, (10_,20_) and (11_,20_), are removed. The outgoing edges from State 01_ are removed because 01_ is not an initial state. States like 20_ are called the *deadend* states. The

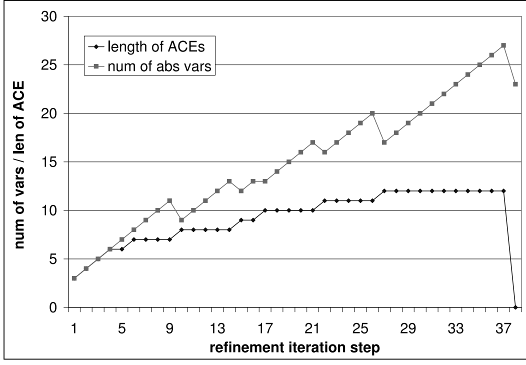

Figure 4: The effect of the generational refinement, with the refinement minimization.

concept of deadend states is critically involved in the refinement variable selection algorithm, as discussed below in Section 5. We must stress that all these splits, which make the SORs change from the one in Part (a) to the one in Part (b), are done before M_2 is brought in.

The second step of refinement is to actually take the composition of M_2 with the remaining edges of the SORs (i.e., those edges in Part (b)). This kills the edges (11_,21_) and (21_,31_), and leads to the reduced SORs in Part (c).

After the above refinement step, we are still at the length-3 generation; the number of length-3 spurious ACEs is decreased, but we have not killed them all. 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 Part (d). The result is a disconnection of I and $\neg p$, since after this reduction there is no outgoing edge from the sole remaining initial state.

At this point, we have proven that there is no CCE of length 3, so this generation of refinements is complete. Note that during these two refinement steps (i.e., adding v_2 and adding v_3), the SORs are updated *incrementally*, that is, inside the existing SORs. The BDD don't cares associated with this incremental process lend critical efficiency to the SORs refinement process.

Next, we rebuild the SORs from scratch, which now are of length 4, as is shown in Part (e). This final set of SORs contains a single ACE, which is concretizable, as discussed above in reference to Fig. 2. So we know the given property fails.

The effect of the generational refinements is illustrated in Fig. 4, which is obtained from a real-world circuit on which the given property is true. The upper curve is the number of state variables in the abstract model at different refinement steps, and the lower curve is the shortest ACE lengths at different refinement steps. A *generation* consists of a number of consecutive refinement steps, all with SORs of the same length. Note that every time the shortest ACE length 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. Our experience shows that this is critical in achieving a high abstraction efficiency.

4. The Algorithms

Let $\{S^0, S^1, \ldots, S^L\}$ be the length-L synchronized onion rings, where S^0 is the set of initial states, S^L consists of states satisfying $\neg p$, and the S^j $(0 < j < L)$ is the set of abstract states on the shortest abstract paths from S^0 to S^L.

```
GRAB(M,Φ) {
1   M̂ = INITIALABSTRACTION(M,Φ)
2   while (1) { //Loop over SORs with different length
3       {S^l} = COMPUTESORS(M̂,Φ)
4       if ( {S^l} is empty )
5           return TRUE
6       CCE = MULTITHREADCONCRETIZATION(M,Φ,{S^l})
7       if (CCE not empty)
8           return (FALSE, CCE)
9       {S^l_R} = {S^l}
10      while (1) { //Loop over refinements for current length
11          M̂ = REFINEABSTRACTION(M̂,{S^l_R})
12          {S^l_R} = REDUCESORS(M̂,{S^l_R},Φ)
13          if ({S^l_R} is empty)
14              break
15      }
16      M̂ = REFINEMENTMINIMIZATION(M̂,{S^l})
    } }

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

Figure 5: Abstraction-Refinement algorithm GRAB.

4.1 The Overall Algorithm of GRAB

The pseudo code of our abstraction and refinement algorithm is given in Fig. 5. We call the algorithm GRAB, for Generational Refinement of Ariadne's Bundle. GRAB accepts as inputs the concrete model M and the property Φ (in this paper, $\Phi = \mathsf{AG}\, p$, where p is a propositional formula over the state variables.)

First, an initial abstract model $\hat{M}$ is created, with only those state variables that appear in the local support of the property. The outer loop is over the length, L, of the current generation of refined SORs. As the abstract model is 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 on $\hat{M}$, it cannot be reached in M either. In this case of early termination, GRAB returns TRUE. Otherwise, we compute the first set of SORs of the current length L.

A SAT-based concretization test of the current the SORs is then conducted on the concrete model. Here, we simultaneously try to concretize all the ACEs present in the SORs by one SAT instance. If any ACE of the current length can be concretized, we have the second case of early termination (Line 8). Thus the property Φ is proved FALSE, and the CCE (Concrete Counter Example) is returned.

Otherwise, we start the inner loop over the refinements for this generation. The length of the SORs (S^l_R stands for the "reduced SORs") does not change in the inner loop, but the number of ACEs in it decreases monotonically. Note that there is no concretization test in the inner loop, for all the ACEs in S^l_R have been proved spurious.

Each time the abstract model is refined (Line 11), the SORs are reduced (Line 12) until all the spurious ACEs in them are removed. Typically a few passes through the inner loop produce the breakout, which implies the current generation of refinement constitutes a sufficient set that kills the entire length-L SORs.

The details of the subprocedure calls at Lines 6 and 16 will be discussed in the following two subsections. We reserve the treatment of REFINEABSTRACTION (Line 11) and the game theoretic heuristic PICKBESTVAR for Section 5.

Prior art in abstraction/refinement algorithms [5, 6, 4] can be described with a similar framework of pseudocode. However, these algorithms are all based on the analysis of a single abstract counterexample. We note that even an optimum refinement based on a single ACE could not necessarily guarantee a good overall refinement, although our experimental results show that sometimes good results can be obtained. We will compare GRAB to these alternative methods in the experimental results section.

4.2 Multi-Thread Concretization Test

The multi-thread concretization test is formulated as a satisfiability problem, which can be solved by the state-of-the-art SAT solvers. For this purpose, the concrete transition relations are unrolled to length L and constrained at each time step by the corresponding SOR. To achieve this, we translate theses SORs into the CNF (Conjunctive Normal Form) format. Let the CNF formula $\Psi = \Psi_M \wedge \Psi_S$, where Ψ_M represents the bounded concrete model, and Ψ_S represets the constraints from the abstract SORs.

$$\Psi_M = I(X^0) \bigwedge_{0 \le l < L} T(X^l, W^l, X^{l+1}) \ ,$$

$$\Psi_S = \bigwedge_{0 \le l \le L} S^l(X^l) \ ,$$

where X^l and W^l are the state variables and inputs at the lth time step. (Ψ_M is constructed by unrolling the concrete model exactly L time steps, and representing the new bounded model with only *and* gates and *inverters*. The *and-inverter* graph is then translated into the CNF format.) Note that this is similar to building a length-L instance in Bounded Model Checking (BMC) [2].

In the BDD-based model checker, each S^l of the SORs is a BDD representation of a set of states at the lth time step. In order to build the CNF formula for $S^l(X^l)$, an *and-inverter* graph must be built for that BDD; in our implementation, this is done by converting each BDD node into a 2-input *multiplexer* (which in turn can be represented by 3 *nand* gates) [10]. Once the *and-inverter* graphs are built, encoding them into CNF is straightforward.

Ψ is satisfiable iff there exists a concrete counter example inside the abstract SORs. If satisfiable, the assignment returned by the SAT solver represents a CCE from an initial state to a $\neg p$ state.

4.3 Refinement Minimization

Given a sufficient set of refinement variables, and the spurious ACE(s), the refinement minimization problem can be defined as finding the minimal subset of refinement variables that can kill the spurious ACE(s). Refinement minimization was first proposed in [22] and then used in [4], where a single ACE was used; the refinement minimization was triggered every time a single ACE was killed.

In our method, however, we do not try to achieve such a local minimality; we conduct the minimization only at the end of each generation, when all the length-L ACEs have been killed. The result is a potentially more global mimimality – the minimal set of refinement variables is with respect to the bundle of ACEs.

Our refinement minimization also uses the SAT solvers, and the satisfiability checks in it is similar to the multi-thread concretization test. For each variable in the sufficient set, we first try to remove it from the set, the SAT solver is then used to check whether the killed ACEs come back. If they do not come back, that variable is proved to be redundant, otherwise, it must be added back. Unlike the multi-thread concretization test, the satisfiability checks here are conducted on the abstract models, so they can be much easier to solve.

5. Game-Theoretic Refinement

In this section we explain the algorithm used in REFINEABSTRACTION, and we do this by first formalizing the refinement problem as a game. The set of *invisible variables*, $\check{x}$, are the free variables (i.e., inputs) in $\hat{M}$. Let $\check{x}$ be partitioned into two sets ($\check{x} = w_E \cup w_S$): w_E are the variables controlled by a hostile *environment* to force the abstract *system* $\hat{M}$ through the spurious ACEs; the remaining variables w_S are controlled by the abstract *system* $\hat{M}$ to play against the hostile *environment*.

Given $\hat{M}$, the two sets w_E, w_S and a target predicate $\neg p$, the model checking of $\mathsf{AG}\, p$ on $\hat{M}$ can be viewed as a two-player concurrent reachability game [8][13]. The positions of the game correspond to the states of $\hat{M}$; the two players are the hostile *environment* and the abstract *system*. From one position $\hat{X}$ ($\hat{X}$ is a valuation of $\hat{x}$; similarly let capital values of other vector names stand for their valuations), 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 hostile *environment* is to go through spurious paths and reach a state labeled $\neg p$ in spite of the abstract *system*'s opposition.

A (memoryless) strategy for the *environment* is a function that maps each state of $\hat{M}$ to one valuation of the variables in w_E. Likewise, a strategy for the *system* is a function that maps each state of $\hat{M}$ to one valuation of the variables in w_S. A position $\hat{X}$ is a winning position for the environment if there exists an *environment* strategy such that, for all *system* strategies, $\neg p$ is eventually satisfied. If $w_E = \check{x}$ (the hostile *environment* controls all the invisible variables) and $\hat{M}$ is deterministic, the *environment* can force the *system* along any spurious ACEs. Note that this is exactly the case in $\hat{M}$ before the refinement.

The refinement problem can be stated as follows: Among all the possible partitions of $\check{x} = w_E \cup w_S$, we choose the one that gives the *environment* the least number of winning positions. Note that in a "MinMax" game without a "win-win strategy", the partition that favors the hostile *environment* the least also favors the abstract *system* most. Then we refine with the variables in w_S by adding their transition bit-relation into the abstract model; this makes them not free variables any more (in the game, they will be controlled by the abstract *system*).

Given a partition $\{w_E, w_S\}$ of the $\check{x}$ variables, and the abstract SORs $\{S^j\}$, the *environment*'s winning positions inside S^j can be represented symbolically as

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

which is the set of states from which the hostile *environment* can force the abstract *system* to S^{j+1} despite its opposition. Note that although universal abstraction is not the same operation as composition, they both reduce the number of enabled edges. Further, it can be shown that when an edge label has a variable which factors out of its label (all the edges in Fig. 6 except the edge from state 5 to state 7), then composing that variable with the abstract model splits the abstract edge into 2 edges (instead of 4). For such edges, a split is created, in which one of the two split nodes has no fanout—that is, it is a *deadend split*.

Let us consider the abstract model in Fig. 6 as an example. Suppose $I = \{1, 2, 5, 6\}$. Then, $S^0 = I = \{1, 2, 5, 6\}$, $S^1 = \{3, 4\}$, $\check{x} = \{g, f\}$. When the partition is such that $w_E = \{g\}$ and $w_S = \{f\}$, the winning position for the *environment* is $\{1, 2\}$. State 1 is a winning position for the *environment* because, when the hostile *environment*

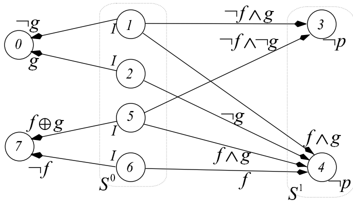

Figure 6: Illustration of the winning position.

chooses the valuation $g = 1$, the abstract *system* will be forced to a $\neg p$ state (either 3 or 4) no matter what its strategy (the value of f) is .

Given the partition $\{w_E, w_S\}$, the normalized number of winning positions for the *environment* inside abstract states S^j can be computed as

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

Here $|.|$ stands for the cardinality (or number of states). For the purpose of refinement, we seek such a partition that, when adding the w_S variables into the abstract model, removes the largest number of spurious edges. N_j, the estimated number of spurious edges that might be killed, is a good indicator of the impact of refining with respect to the w_S variables. Thus we prefer the partition that gives N_j the lowest value.

According to the definition of N_j and the given example in Fig. 6,

$$N_0^{\{\{g,f\},\{\}\}} = 1.0,$$
$$N_0^{\{\{g\},\{f\}\}} = 0.5,$$
$$N_0^{\{\{f\},\{g\}\}} = 0.25,$$
$$N_0^{\{\{\},\{g,f\}\}} = 0.0.$$

This indicates that g is a better candidate than f for the refinement, because putting g alone in w_S gives the hostile *environment* one winning position, while putting f alone in w_S gives it two winning positions.

Our refinement goal is to select a small set of invisible variables into w_S such that the partition $\{w_E, w_S\}$ minimizes the

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

This is greedily approximated inside REFINEABSTRACTION: The one variable that minimizes the above number is repeatedly picked (Line 19 in Fig. 5). In any case, those w_S variables, together with their transition bit-relation, will be put into our refined model.

The computation of $\exists w_E.\forall w_S.\exists \hat{y}.[S^j(\hat{x}) \wedge \hat{T}(\hat{x},\check{x},\hat{y}) \wedge S^{j+1}(\hat{y})]$ can be made more efficient by pulling S^j out of the quantifications, and by sharing the common intermediate result $\exists \hat{y}.[\hat{T}(\hat{x},\check{x},\hat{y}) \wedge S^{j+1}(\hat{y})]$. We also point out that, the subset $\{w_E, w_S\}$ contains only invisible variables that are in the local support of the current abstract model.

6. Experiments

We have implemented the GRAB algorithm and two competing refinement algorithms in VIS-2.0 [3, 21]. We use CUDD for the BDD-based computation, and Chaff [19] 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, 2 GB of RAM. CPU times are in seconds and are all-inclusive.

Table 1 compares two variants of the GRAB algorithm against the BDD-based invariant checking algorithm in VIS (CI), Bounded Model Checking (BMC), the SepSet algorithm [6], a variant of SepSet called SepSet+, and the conflict analysis algorithm of [4]. 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 inductive invariants, but none of our invariants is 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 hours. 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 [21].

In Table 1, 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 counter example, 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; **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**, 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, because they use the SAT solver (instead of BDDs) to compute the refinement; unlike GRAB, they do not need backward reachability analysis. BMC uses no BDDs.

In the I12-p1 instance, the **sat** time for GRAB is markedly higher than elsewhere. This is because I12-p1 is a model inheriantly hard for BMC/SAT; it is a failing property, and BMC can not solve it within 8 hours.

Table 2 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 1.

Table 2 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 10 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 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 [6, 4], there remain differences between the originals and our rewritings. For instance, our current implementation, considers the bit relation of one state variable as an atom: When a variable becomes part of the abstract model, all the state variables in its support become inputs to the model. This is not the case of the original methods of [6, 4], and will in some cases impede the search for a

Table 1: 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
I12-p1	119	370	>8 h	>8 h	6202	26	31	6062	26	31	>8 h	(3)	(61)	3025	15	20	**2503**	30	16	1382

good abstraction. However, the drawback is shared by all methods we implemented, and therefore should not have a major impact on the comparison we present.

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 non-trivial abstract model. (It requires at least one refinement iteration.) The abstraction efficiency is 0 (100%) if the final model contains all (no) state variables. Fig. 7 shows scatterplots of the abstraction efficiency of SepSet, CA, and GRAB. SepSet+ behaves like SepSet. Each point below the diagonal represents a win for GRAB.

Table 2: The correlation between the final proofs (GRAB vs. 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)	?	?	?	?	?
I12-p1	119	370	16	(61)	?	?	?	?	?

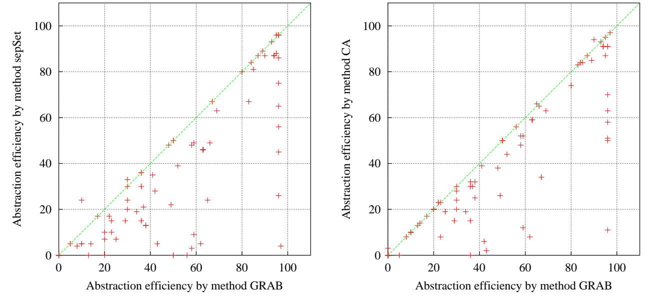

Figure 7: Comparison of the abstraction efficiency: (1) GRAB vs. SepSet; (2) GRAB vs. CA.

Scatterplots 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.

7. Conclusions

Recent abstraction refinement research and advances in 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 characterizes those methods raises the issue of global versus local guidance in the search for counter examples.

In this paper we have shown that significant performance im-

provements can be achieved by emphasizing global guidance. For a given invariant, our approach analyzes all counter examples of the shortest length at once. This leads to higher abstraction efficiency relative to methods that base the refinement on the analysis of one counter example only. Our approach to refinement is scalable in the sense that the computation of the refinement only requires the examination of the abstract model. We still need the concrete model to check whether abstract counter examples are spurious, but in our experiments the cost of concretizing multiple paths was usually less than the cost of SAT-based refinement procedures. A practical lessening of the concretization check problem may also come from an incremental approach like the one of [1].

Our current work aims at exploring further mechanisms for global guidance, in particular with regard to the trade-off between cost and predictive power, and the bias between trying to prove or disprove a property. The granularity of the refinement has also great impact on performance, and our efforts are directed at providing more control over this parameter.

Future work includes widening the locality scope of the set of refinement candidates, which is currently limited to the immediate support of the current abstraction. We have noted cases in which variables that are in the support of the local support give a better refinement due to increased deadend split production. We are also considering making the definition of abstraction efficiency more precise, to distinguish between truly successful refinement algorithms, and algorithms that repeatedly pick many poor variables, and then rely on refinement minimization. In some cases we have studied, the SAT-based conflict analysis method falls into this latter category.

References

[1] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 65–77. Springer-Verlag, July 2002. LNCS 2404.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] 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 M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, Nov. 2002. LNCS 2517.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 154–169. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[6] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[8] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, Oct. 1991.

[9] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 176–191, Warsaw, Poland, Apr. 2003. LNCS 2619.

[10] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Abstraction and BDDs complement SAT-based BMC in DiVer. In W. A. Hunt and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 206–209. Springer-Verlag, Berlin, 2003. LNCS 2725.

[11] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 423–427. Springer-Verlag, Berlin, 1996. LNCS 1102.

[12] J.-Y. Jang, I.-H. Moon, and G. D. Hachtel. Iterative abstraction-based CTL model checking. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE00)*, pages 502–507, Paris, France, Mar. 2000.

[13] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 445–459, Grenoble, France, Apr. 2002. LNCS 2280.

[14] P. Kelb, D. Dams, and R. Gerth. Practical symbolic model checking of the full μ-calculus using compositional abstractions. Technical Report 95-31, Department of Computing Science, Eindhoven University of Technology, 1995.

[15] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[16] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[17] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, San Jose, CA, Nov. 1996.

[18] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, Apr. 2003. LNCS 2619.

[19] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[20] A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 12–23. Springer-Verlag, Berlin, 1997. LNCS 1254.

[21] URL: http://vlsi.colorado.edu/∼vis.

[22] 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 *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.