Counterexample-guided Abstraction Refinement *

Edmund Clarke[1], Orna Grumberg[2], Somesh Jha[1], Yuan Lu[1], and Helmut Veith[1,3]

[1] Carnegie Mellon University, Pittsburgh, USA [2] Technion, Haifa, Israel
[3] Vienna University of Technology, Austria

Abstract. We present an automatic iterative abstraction-refinement methodology in which the initial abstract model is generated by an automatic analysis of the control structures in the program to be verified. Abstract models may admit erroneous (or "spurious") counterexamples. We devise new symbolic techniques which analyze such counterexamples and refine the abstract model correspondingly. The refinement algorithm keeps the size of the abstract state space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. We describe an implementation of our methodology in NuSMV. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code confirm the effectiveness of our approach.

1 Introduction

The state explosion problem remains a major hurdle in applying model checking to large industrial designs. Abstraction is certainly the most important technique for handling this problem. In fact, it is essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, often requiring considerable creativity. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions. In this paper, we describe an automatic abstraction technique for ACTL* specifications which is based on an analysis of the structure of formulas appearing in the program ($\mathrm{ACTL}^\star$ is a fragment of $\mathrm{CTL}^\star$ which only allows universal quantification over paths). In general, our technique computes an upper approximation of the original program. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. The main contribution of this paper is an efficient automatic refinement technique which uses information obtained from erroneous counterexamples. The refinement algorithm keeps the size of the abstract state

* This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Max Kade Foundation. One of the authors is also supported by Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code confirm the competitiveness of our implementation. Although our current implementation is based on NuSMV, it is in principle not limited to the input language of SMV and can be applied to other languages.

Our paper follows the general framework established by Clarke, Grumberg, and Long [10]. We assume that the reader has some familiarity with that framework. In our methodology, *atomic formulas* are automatically extracted from the program that describes the model. The atomic formulas are similar to the *predicates* used for abstraction by Graf and Saidi [13] and later in [11, 20]. However, instead of using the atomic formulas to generate an abstract global transition system, we use them to construct an explicit *abstraction function*. The abstraction function preserves logical relationships among the atomic formulas instead of treating them as independent propositions. The initial abstract model is constructed by adapting the *existential abstraction* techniques proposed in [8, 10] to our framework. Then, a traditional model checker is used to determine whether $\mathrm{ACTL}^\star$ properties hold in the abstract model. If the answer is yes, then the concrete model also satisfies the property. If the answer is no, then the model checker generates a counterexample. Since the abstract model has more behaviors than the concrete one, the abstract counterexample might not be valid. We say that such a counterexample is *spurious*. Such abstraction techniques are also known as false negative techniques.

In our methodology, we provide a new symbolic algorithm to determine whether an abstract counterexample is spurious. If the counterexample is not spurious, we report it to the user and stop. If the counterexample is spurious, the abstraction function must be refined to eliminate it. In our methodology, we identify the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model. The last abstract state in this prefix is split into less abstract states so that the spurious counterexample is eliminated. Thus, a more refined abstraction function is obtained. Note that there may be many ways of splitting the abstract state; each determines a different refinement of the abstraction function. It is desirable to obtain the coarsest refinement which eliminates the counterexample because this corresponds to the *smallest* abstract model that is suitable for verification. We prove, however, that finding the coarsest refinement is NP-hard. Because of this, we use a polynomial-time algorithm which gives a suboptimal but sufficiently good refinement of the abstraction function. The applicability of our heuristic algorithm is confirmed by our experiments. Using the refined abstraction function obtained in this manner, a new abstract model is built and the entire process is repeated. Our methodology is complete for the fragment of $\mathrm{ACTL}^\star$ which has counterexamples that are either paths or loops, i.e., we are guaranteed to either find a valid counterexample or prove that the system satisfies the desired property. In principle, our methodology can be extended to all of $\mathrm{ACTL}^\star$.

Using counterexamples to refine abstract models has been investigated by a number of other researchers beginning with the *localization reduction* of Kurshan [14]. He models a concurrent system as a composition of L-processes $L_1, \ldots, L_n$ (L-processes are described in detail in [14]). The localization reduction is an iterative technique that starts with a small subset of relevant L-processes that are topologically close to the specification in the *variable dependency graph*. All other program variables are abstracted away with nondeterministic assignments. If the counterexample is found to be spurious, additional variables are added to eliminate the counterexample. The heuristic for selecting these variables also uses information from the variable dependency graph. Note that the localization reduction either leaves a variable unchanged or replaces it by a nondeterministic assignment. A similar approach has been described by Balarin in [2, 15]. In our approach, the abstraction functions exploit logical relationships among variables appearing in atomic formulas that occur in the control structure of the program. Moreover, the way we use abstraction functions makes it possible to distinguish many degrees of abstraction for each variable. Therefore, in the refinement step only very small and local changes to the abstraction functions are necessary and the abstract model remains comparatively small.

Another refinement technique has recently been proposed by Lind-Nielson and Andersen [17]. Their model checker uses upper and lower approximations in order to handle all of CTL. Their approximation techniques enable them to avoid rechecking the entire model after each refinement step while guaranteeing completeness. As in [2, 14] the variable dependency graph is used both to obtain the initial abstraction and in the refinement process. Variable abstraction is also performed in a similar manner. Therefore, our abstraction-refinement methodology relates to their technique in essentially the same way as it relates to the classical localization reduction.

A number of other papers [16, 18, 19] have proposed abstraction-refinement techniques for CTL model checking. However, these papers do not use counterexamples to refine the abstraction. We believe that the methods described in these papers are orthogonal to our technique and may even be combined with ours in order to achieve better performance. A recent technique proposed by Govindaraju and Dill [12] may be a starting point in this direction, since it also tries to identify the first spurious state in an abstract counterexample. It randomly chooses a concrete state corresponding to the first spurious state and tries to construct a real counterexample starting with the image of this state under the transition relation. The paper only talks about safety properties and path counterexamples. It does not describe how to check liveness properties with cyclic counterexamples. Furthermore, our method does not use random choice to extend the counterexample; instead it analyzes the cause of the spurious counterexample and uses this information to guide the refinement process. A more detailed comparison with related work will be given in the full version

Summarizing, our technique has a number of advantages over previous work:

(i) The technique is complete for an important fragment of ACTL*.

(ii) The initial abstraction and the refinement steps are efficient and entirely automatic. All algorithms are symbolic.

(iii) In comparison to methods like the localization reduction, we distinguish more degrees of abstraction for each variable. Thus, the changes in the refinement are potentially finer in our approach.

(iv) The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small.

We have implemented our new methodology in NuSMV [6] and applied it to a number of benchmark designs [6]. In addition we have used it to debug a large IP core being developed at Fujitsu [1]. The design has about 350 symbolic variables which correspond to about 500 latches. Before using our methodology, we implemented the *cone of influence* reduction [8] in NuSMV to enhance its ability to check large models. Neither our enhanced version of NuSMV nor the recent version of SMV developed by Yang [23] were able to verify the Fujitsu IP core design. However, by using our new technique, we were able to find a subtle error in the design. Our program automatically abstracted 144 symbolic variables and performed three refinement steps. Currently, we are evaluating the methodology on other complex industrial designs.

The paper is organized as follows: Section 2 gives the basic definitions and terminology used throughout the paper. A general overview of our methodology is given in Section 3. Detailed descriptions of our abstraction-refinement algorithms are provided in Section 4. Performance improvements for the implementation are described in Section 5. Experimental results are presented in Section 6. Future research is discussed in Section 7.

2 Preliminaries

A *program* P has a finite set of variables $V = \{v_1, \cdots, v_n\}$, where each variable v_i has an associated finite domain D_{v_i}. The set of all possible states for program P is $D_{v_1} \times \cdots D_{v_n}$ which we denote by D. *Expressions* are built from variables in V, constants in D_{v_i}, and function symbols in the usual way, e.g. $v_1 + 3$. *Atomic formulas* are constructed from expressions and relation symbols, e.g. $v_1 + 3 < 5$. Similarly, *predicates* are composed of atomic formulas using negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). Given a predicate p, $\mathrm{Atoms}(p)$ is the set of atomic formulas occurring in it. Let p be a predicate containing variables from V, and $d = (d_1, \ldots, d_n)$ be an element from D. Then we write $d \models p$ when the predicate obtained by replacing each occurrence of the variable v_i in p by the constant d_i evaluates to true.

Each variable v_i in the program has an associated *transition block*, which defines both the initial value and the transition relation for the variable v_i. An example of a transition block for the variable v_i is shown in Figure 1, where $I_i \subseteq D_{v_i}$ is the initial

init(v_i) := I_i;	**init**(x) := 0;	**init**(y) := 1;
next(v_i) := **case**	**next**(x) := **case**	**next**(y) := **case**
$C_i^1 : A_i^1$;	$reset = \mathrm{TRUE} : 0$;	$reset = \mathrm{TRUE} : 0$;
$C_i^2 : A_i^2$;	$x < y : x + 1$;	$(x = y) \wedge \neg(y = 2) : y + 1$;
$\cdots : \cdots$;	$x = y : 0$;	$(x = y) : 0$;
$C_i^k : A_i^k$;	**else** : x;	**else** : y;
esac;	**esac**;	**esac**;

Fig. 1. A generic transition block and a typical example

expression for the variable v_i, each condition C_i^j is a predicate, and A_i^j is an expression.

The semantics of the transition block is similar to the semantics of the **case** statement in the modeling language of SMV, i.e., find the least j such that in the current state condition C_i^j is true and assign the value of the expression A_i^j to the variable v_i in the next state.

We assume that the specifications are written in a fragment of $\mathrm{CTL}^\star$ called $\mathrm{ACTL}^\star$ (see [10]). Assume that we are given an $\mathrm{ACTL}^\star$ specification φ, and a program P. For each transition block B_i let $\mathrm{Atoms}(B_i)$ be the set of atomic formulas that appear in the conditions. Let $\mathrm{Atoms}(\varphi)$ be the set of atomic formulas appearing in the specification φ. $\mathrm{Atoms}(P)$ is the set of atomic formulas that appear in the specification or in the conditions of the transition blocks.

Each program P naturally corresponds to a labeled *Kripke structure* $M = (S, I, R, L)$, where $S = D$ is the set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{\mathrm{Atoms}(P)}$ is a labelling given by $L(d) = \{f \in \mathrm{Atoms}(P) \mid d \models f\}$. Translating a program into a Kripke structure is straightforward and will not be described here.

An abstraction h for a program P is given by a surjection $h : D \to \widehat{D}$. Notice that the surjection h induces an equivalence relation $\equiv$ on the domain D in the following manner: let d, e be states in D, then

$$d \equiv e \text{ iff } h(d) = h(e).$$

Since an abstraction can be represented either by a surjection h or by an equivalence relation $\equiv$, we sometimes switch between these representations to avoid notational overhead.

Assume that we are given a program P and an abstraction function h for P. The *abstract Kripke structure* $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L})$ corresponding to the abstraction function h is defined as follows:

1. $\widehat{S}$ is the abstract domain $\widehat{D}$.
2. $\widehat{I}(\widehat{d})$ iff $\exists d(h(d) = \widehat{d} \wedge I(d))$.
3. $\widehat{R}(\widehat{d_1}, \widehat{d_2})$ iff $\exists d_1 \exists d_2 (h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$.
4. $\widehat{L}(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$. (This definition will be justified in Theorem 1.)

This abstraction technique is called *existential abstraction* [8]. An atomic formula f *respects* an abstraction function h if for all d and d' in the domain D, $(d \equiv d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. Let $\widehat{d}$ be an abstract state. $\widehat{L}(\widehat{d})$ is *consistent*, if all concrete states corresponding to $\widehat{d}$ satisfy all labels in $\widehat{L}(\widehat{d})$, i.e., for all $d \in h^{-1}(\widehat{d})$ it holds that $d \models \bigwedge_{f \in \widehat{L}(\widehat{d})} f$.

Theorem 1. *Let h be an abstraction and φ be an $\mathrm{ACTL}^\star$ specification where the atomic subformulas respect h. Then the following holds: (i) $\widehat{L}(\widehat{d})$ is consistent for all abstract states $\widehat{d}$ in $\widehat{M}$; (ii) $\widehat{M} \models \varphi \Rightarrow M \models \varphi$.*

In other words, correctness of the abstract model implies correctness of the concrete model. On the other hand, if the abstract model invalidates an $\mathrm{ACTL}^\star$ specification, i.e., $\widehat{M} \not\models \varphi$, *the actual model may still satisfy the specification.*

Example 1. Assume that for a traffic light controller (see Figure 2), we want to prove $\psi = \mathbf{AG\,AF}(state = red)$ using the abstraction function $h(red) = red$ and $h(green) = h(yellow) = go$. It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite trace $\langle red, go, go, \ldots \rangle$ that invalidates the specification.

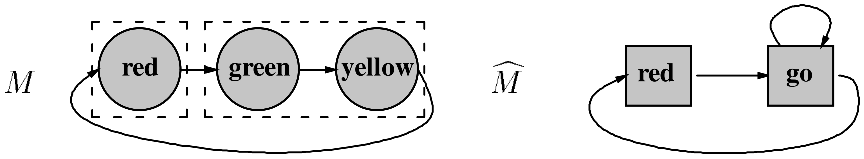

Fig. 2. Abstraction of a Traffic Light.

If an abstract counterexample does not correspond to some concrete counterexample, we call it spurious. For example, $\langle red, go, go, \ldots \rangle$ in the above example is a spurious counterexample.

When the set of possible states is given as the product $D_1 \times \cdots D_n$ of smaller domains, an abstraction h can be described by surjections $h_i : D_i \to \widehat{D_i}$, such that $h(d_1, \ldots, d_n)$ is equal to $(h_1(d_1), \ldots, h_n(d_n))$, and $\widehat{D}$ is equal to $\widehat{D_1} \times \cdots \widehat{D_n}$. In this case, we write $h = (h_1, \ldots, h_n)$. The equivalence relations $\equiv_i$ corresponding to the individual surjections h_i induce an equivalence relation $\equiv$ over the entire domain $D = D_1 \times \cdots \times D_n$ in the obvious manner:

$$(d_1, \cdots, d_n) \equiv (e_1, \cdots, e_n) \text{ iff } d_1 \equiv_1 e_1 \wedge \cdots \wedge d_n \equiv_n e_n$$

In previous work on existential abstraction [10], abstractions were defined for each variable domain, i.e., D_i in the above paragraph was chosen to be D_{v_i}, where D_{v_i} is the set of possible values for variable v_i. Unfortunately, many abstraction functions h can not be described in this simple manner. For example, let $D = \{0,1,2\} \times \{0,1,2\}$, and $\widehat{D} = \{0,1\} \times \{0,1\}$. Then there are $4^9 = 262144$ functions h from D to $\widehat{D}$. Next, consider $h = (h_1, h_2)$. Since there are $2^3 = 8$ functions from $\{0,1,2\}$ to $\{0,1\}$, there are only 64 functions of this form from D to $\widehat{D}$.

In this paper, we define abstraction functions in a different way. We partition the set V of variables into sets of related variables called *variable clusters* $VC_1, \ldots, VC_m$, where each variable cluster VC_i has an associated domain $D_{VC_i} := \prod_{v \in VC_i} D_v$. Consequently, $D = D_{VC_1} \times \cdots D_{VC_m}$. We define abstraction functions as surjections on the domains D_{VC_i}, i.e., D_i in the above paragraph is equal to D_{VC_i}. Thus, the notion of abstraction used in this paper is more general than the one used in [10].

3 Overview

For a program P and an $\mathrm{ACTL}^\star$ formula φ, our goal is to check whether the Kripke structure M corresponding to P satisfies φ. Our methodology consists of the following steps.

1. *Generate the initial abstraction:* We generate an initial abstraction h by examining the transition blocks corresponding to the variables of the program. We consider the conditions used in the **case** statements and construct variable clusters for variables which interfere with each other via these conditions. Details can be found in Section 4.1.
2. *Model-check the abstract structure:* Let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction h. We check whether $\widehat{M} \models \varphi$. If the check is affirmative, then we can conclude that $M \models \varphi$ (see Theorem 1). Suppose the check reveals that there is a counterexample $\widehat{T}$. We ascertain whether $\widehat{T}$ is an actual counterexample, i.e., a counterexample in the unabstracted structure M. If $\widehat{T}$ turns out to be an actual counterexample, we report it to the user, otherwise $\widehat{T}$ is a spurious counterexample, and we proceed to step 3.
3. *Refine the abstraction:* We refine the abstraction function h by partitioning a *single equivalence class* of $\equiv$ so that after the refinement the abstract structure $\widehat{M}$ corresponding to the refined abstraction function does not admit the spurious counterexample $\widehat{T}$. We will discuss partitioning algorithms for this purpose in Section 4.3. After refining the abstraction function, we return to step 2.

4 The Abstraction-Refinement Framework

4.1 Generating the Initial Abstraction

Assume that we are given a program P with n variables $\{v_1, \cdots, v_n\}$. Given an atomic formula f, let $var(f)$ be the set of variables appearing in f, e.g., $var(x = y)$ is $\{x, y\}$. Given a set of atomic formulas U, $var(U)$ equals $\bigcup_{f \in U} var(f)$. In general, for any syntactic entity X, $var(X)$ will be the set of variables appearing in X. We say that two atomic formulas f_1 and f_2 *interfere* iff $var(f_1) \cap var(f_2) \neq \emptyset$. Let $\equiv_I$ be the equivalence relation on $\mathrm{Atoms}(P)$ that is the reflexive, transitive closure of the interference relation. The equivalence class of an atomic formula $f \in \mathrm{Atoms}(P)$ is called the *formula cluster* of f and is denoted by $[f]$. Let f_1 and f_2 be two atomic formulas. Then $var(f_1) \cap var(f_2) \neq \emptyset$ implies that $[f_1] = [f_2]$. In other words, a variable v_i cannot appear in formulas that belong to two different formula clusters. Moreover, the formula clusters induce an equivalence relation $\equiv_V$ on the set of variables V in the following way:

> $v_i \equiv_V v_j$ if and only if v_i and v_j appear in atomic formulas that belong to the same formula cluster.

The equivalence classes of $\equiv_V$ are called *variable clusters*. For instance, consider a formula cluster $FC_i = \{v_1 > 3, v_1 = v_2\}$. The corresponding variable cluster is $VC_i = \{v_1, v_2\}$. Let $\{FC_1, \ldots, FC_m\}$ be the set of formula clusters and $\{VC_1, \ldots, VC_m\}$ the set of corresponding variable clusters. We construct the initial abstraction $h = (h_1, \ldots, h_m)$ as follows. For each h_i, we set $D_{VC_i} = \prod_{v \in VC_i} D_v$, i.e., D_{VC_i} is the domain corresponding to the variable cluster VC_i. Since the variable clusters form a partition of the set of variables V, it follows that $D = D_{VC_1} \times \cdots D_{VC_m}$. For each variable cluster $VC_i = \{v_{i_1}, \ldots, v_{i_k}\}$, the corresponding abstraction h_i is defined on D_{VC_i} as follows.

$h_i(d_1, \cdots, d_k) = h_i(e_1, \cdots, e_k)$ iff for all atomic formulas $f \in FC_i$, $(d_1, \cdots, d_k) \models f \Leftrightarrow (e_1, \cdots, e_k) \models f$.

In other words two values are in the same equivalence class if they cannot be "distinguished" by atomic formulas appearing in the formula cluster FC_i. The following example illustrates how we construct the initial abstraction h.

Example 2. Consider the program P with three variables $x, y \in \{0, 1, 2\}$, and $reset \in \{\mathrm{TRUE}, \mathrm{FALSE}\}$ shown in Figure 1. The set of atomic formulas is $\mathrm{Atoms}(P) = \{(reset = \mathrm{TRUE}), (x = y), (x < y), (y = 2)\}$. There are two formula clusters, $FC_1 = \{(x = y), (x < y), (y = 2)\}$ and $FC_2 = \{(reset = \mathrm{TRUE})\}$. The corresponding variable clusters are $\{x, y\}$ and $\{reset\}$, respectively. Consider the formula cluster FC_1. Values $(0, 0)$ and $(1, 1)$ are in the same equivalence class because for all the atomic formulas f in the formula cluster FC_1 it holds that $(0, 0) \models f$ iff $(1, 1) \models f$. It can be shown that the domain $\{0, 1, 2\} \times \{0, 1, 2\}$ is partitioned into a total of five equivalence classes by this criterion. We denote these classes by the natural numbers $0, 1, 2, 3, 4$, and list them below:

$$\begin{aligned}
0 &= \{(0, 0), (1, 1)\}, \\
1 &= \{(0, 1)\}, \\
2 &= \{(0, 2), (1, 2)\}, \\
3 &= \{(1, 0), (2, 0), (2, 1)\}, \\
4 &= \{(2, 2)\}
\end{aligned}$$

The domain $\{\mathrm{TRUE}, \mathrm{FALSE}\}$ has two equivalence classes – one containing FALSE and the other TRUE. Therefore, we define two abstraction functions $h_1 : \{0, 1, 2\}^2 \rightarrow \{0, 1, 2, 3, 4\}$ and $h_2 : \{\mathrm{TRUE}, \mathrm{FALSE}\} \rightarrow \{\mathrm{TRUE}, \mathrm{FALSE}\}$. The first function h_1 is given by $h_1(0, 0) = h_1(1, 1) = 0$, $h_1(0, 1) = 1$, $h_1(0, 2) = h_1(1, 2) = 2$, $h_1(1, 0) = h_1(2, 0) = h_1(2, 1) = 3$, $h_1(2, 2) = 4$. The second function h_2 is just the identity function, i.e., $h_2(reset) = reset$. Given the abstraction functions, we use the standard existential abstraction techniques to compute the abstract model.

4.2 Model Checking the Abstract Model

Given an $\mathrm{ACTL}^\star$ specification φ, an abstraction function h (assume that φ respects h), and a program P with a finite set of variables $V = \{v_1, \cdots, v_n\}$, let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction function h. We use standard symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the specification φ. If it does, then by Theorem 1 we can conclude that the original Kripke structure also satisfies φ. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the rest of this section, we will focus on counterexamples which are either *(finite) paths* or *loops*.

Identification of Spurious Path Counterexamples First, we will tackle the case when the counterexample $\widehat{T}$ is a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. Given an abstract state $\widehat{s}$, the set of concrete states s such that $h(s) = \widehat{s}$ is denoted by $h^{-1}(\widehat{s})$, i.e., $h^{-1}(\widehat{s}) = \{s | h(s) = \widehat{s}\}$.

We extend h^{-1} to sequences in the following way: $h^{-1}(\widehat{T})$ is the set of concrete paths given by the following expression

$$\{\langle s_1, \cdots, s_n \rangle | \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1})\}.$$

We will occasionally write h^{-1}_{path} to emphasize the fact that h^{-1} is applied to a sequence. Next, we give a *symbolic* algorithm to compute $h^{-1}(\widehat{T})$. Let $S_1 = h^{-1}(\widehat{s_1}) \cap I$ and R be the transition relation corresponding to the unabstracted Kripke structure M. For $1 < i \leq n$, we define S_i in the following manner: $S_i := Img(S_{i-1}, R) \cap h^{-1}(\widehat{s_i})$. In the definition of S_i, $Img(S_{i-1}, R)$ is the forward image of S_{i-1} with respect to the transition relation R. The sequence of sets S_i is computed symbolically using OBDDs and the standard image computation algorithm. The following lemma establishes the correctness of this procedure.

Lemma 1. *The following are equivalent:*

(i) The path $\widehat{T}$ corresponds to a concrete counterexample.
(ii) The set of concrete paths $h^{-1}(\widehat{T})$ is non-empty.
(iii) For all $1 \leq i \leq n$, $S_i \neq \emptyset$.

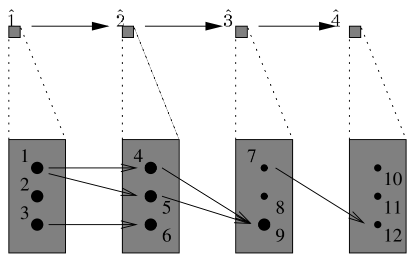

Fig. 3. An abstract counterexample

Algorithm SplitPATH
$S := h^{-1}(\widehat{s_1}) \cap I$
$j := 1$
while ($S \neq \emptyset$ and $j < n$) {
    $j := j + 1$
    $S_{\mathrm{prev}} := S$
    $S := Img(S, R) \cap h^{-1}(\widehat{s_j})$ }
if $S \neq \emptyset$ **then** output counterexample
else output j, S_{prev}

Fig. 4. SplitPATH checks spurious path.

Example 3. Consider a program with only one variable with domain $D = \{1, \cdots, 12\}$. Assume that the abstraction function h maps $x \in D$ to $\lfloor (x-1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7, 8, 9\}$, and $\{10, 11, 12\}$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 3; small dots denote non-reachable states. Suppose that we obtain an abstract counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 1, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that S_4 and therefore $Img(S_3, R)$ are both empty.

It follows from Lemma 1 that if $h^{-1}(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal i ($2 \leq i \leq n$) such that $S_i = \emptyset$. The symbolic Algorithm **SplitPATH** in Figure 4 computes this number and the set of states in

S_{i-1}. In this case, we proceed to the refinement step (see Section 4.3). On the other hand, if the conditions stated in Lemma 1 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

Identification of Spurious Loop Counterexamples Now we consider the case when the counterexample $\widehat{T}$ includes a loop, which we write as $\langle \widehat{s_1}, \cdots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \cdots, \widehat{s_n} \rangle^{\omega}$. The loop starts at the abstract state $\widehat{s_{i+1}}$ and ends at $\widehat{s_n}$. Since this case is more complicated than the path counterexamples, we first present an example in which some of the typical situations occur.

Example 4. We consider a loop $\langle \widehat{s_1} \rangle \langle \widehat{s_2}, \widehat{s_3} \rangle^{\omega}$ as shown in Figure 5. In order to find out if the abstract loop corresponds to concrete loops, we unwind the counterexample as demonstrated in the figure. There are two situations where cycles occur. In the figure,

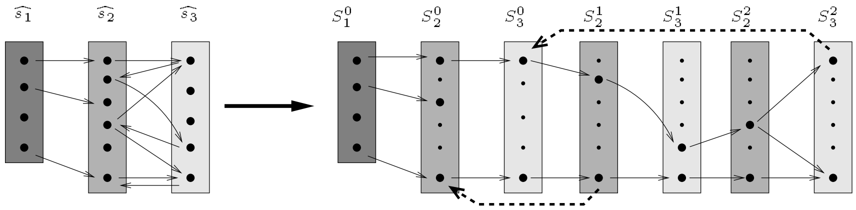

Fig. 5. A loop counterexample, and its unwinding.

for each of these situations, an example cycle (the first one occurring) is indicated by a fat dashed arrow. We make the following important observations: (i) A given abstract loop may correspond to several concrete loops of *different size*. (ii) Each of these loops may start at different stages of the unwinding. (iii) The unwinding eventually becomes periodic (in our case $S_3^0 = S_3^2$), but only after several stages of the unwinding. The size of the period is the least common multiple of the size of the individual loops, and thus, in general *exponential*.

We conclude from the example that a naive algorithm may have exponential time complexity due to an exponential number of loop unwindings. The following surprising result shows that for $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \cdots, \widehat{s_n} \rangle^{\omega}$, the number of unwindings can be bounded by $min = \min_{i+1 \leq j \leq n} |h^{-1}(\widehat{s_j})|$, i.e., the number of unwindings is at most the number of concrete states for any abstract state in the loop. Let $\widehat{T}_{\text{unwind}}$ denote this unwinded loop counterexample, i.e., the finite abstract path $\langle \widehat{s_1}, \ldots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \ldots, \widehat{s_n} \rangle^{min}$. Then the following theorem holds.

Theorem 2. *The following are equivalent: (i) $\widehat{T}$ corresponds to a concrete counterexample. (ii) $h_{\text{path}}^{-1}(\widehat{T}_{\text{unwind}})$ is not empty.*

It can be seen from Example 4 that loop counterexamples are combinatorially more complicated than path counterexamples. Therefore, the proof of Theorem 2 is not immediate; for details, we refer to [7]. We conclude from Theorem 2 that the Algorithm

SplitPATH can be used to analyze abstract loop counterexamples with minor modifications. For easy reference we shall refer to this algorithm as **SplitLOOP**.

4.3 Refining the Abstraction

First, we will consider the case when the counterexample $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$ is a path. Since $\widehat{T}$ does not correspond to a real counterexample, by Lemma 1 (iii) there exists a set $S_i \subseteq h^{-1}(\widehat{s_i})$ with $1 \leq i < n$ such that $Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}}) = \emptyset$ and S_i is reachable from initial state set $h^{-1}(\widehat{s_1}) \cap I$. Since there is a transition from $\widehat{s_i}$ to $\widehat{s_{i+1}}$ in the abstract model, there is at least one transition from a state in $h^{-1}(\widehat{s_i})$ to a state in $h^{-1}(\widehat{s_{i+1}})$ even though there is no transition from S_i to $h^{-1}(\widehat{s_{i+1}})$. We partition $h^{-1}(\widehat{s_i})$ into three subsets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$ as follows (compare Figure 6):

$$\begin{aligned} S_{i,0} &= S_i \\ S_{i,1} &= \{s \in h^{-1}(\widehat{s_i}) | \exists s' \in h^{-1}(\widehat{s_{i+1}}). R(s, s')\} \\ S_{i,x} &= h^{-1}(\widehat{s_i}) \setminus (S_{i,0} \cup S_{i,1}). \end{aligned}$$

Intuitively, $S_{i,0}$ denotes the set of states in $h^{-1}(\widehat{s_i})$ that are reachable from initial states. $S_{i,1}$ denotes the set of states in $h^{-1}(\widehat{s_i})$ that are not reachable from initial states, but have at least one transition to some state in $h^{-1}(\widehat{s_{i+1}})$. The set $S_{i,1}$ cannot be empty since we know that there is a transition from $h^{-1}(\widehat{s_i})$ to $h^{-1}(\widehat{s_{i+1}})$. $S_{i,x}$ denotes the set of states that are not reachable from initial states, and do not have a transition to a state in $h^{-1}(\widehat{s_{i+1}})$. For illustration, consider again the example in Figure 3. Note that $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Using the notation introduced above, we have $S_{3,0} = \{9\}$, $S_{3,1} = \{7\}$, and $S_{3,x} = \{8\}$. Since $S_{i,1}$ is not empty, there is a spurious transition $\widehat{s_i} \to \widehat{s_{i+1}}$. This causes the spurious counterexample $\widehat{T}$. Hence in order to refine the abstraction h so that the new model does not allow $\widehat{T}$, we need a refined abstraction function which separates the two sets $S_{i,0}$ and $S_{i,1}$, i.e., we need an abstraction function, in which no abstract state simultaneously contains states from $S_{i,0}$ and from $S_{i,1}$.

It is natural to describe the needed refinement in terms of equivalence relations: Recall that $h^{-1}(\widehat{s})$ is an equivalence class of $\equiv$ which has the form $E_1 \times \cdots \times E_m$, where each E_i is an equivalence class of $\equiv_i$. Thus, the refinement $\equiv'$ of $\equiv$ is obtained by partitioning the equivalence classes E_j into subclasses, which amounts to refining the equivalence relations $\equiv_j$. The *size of the refinement* is the number of new equivalence classes. Ideally, we would like to find the coarsest refinement that separates the two sets, i.e., the separating refinement with the smallest size. We can show however that this is computationally intractable.

Theorem 3. *(i) The problem of finding the coarsest refinement is NP-hard; (ii) when $S_{i,x} = \emptyset$, the problem can be solved in polynomial time.*

We find that the previously known poblem PARTITION INTO CLIQUES can be reduced to the coarsest refinement problem. The proof is omitted due to space restrictions. On the other hand, we describe a polynomial time algorithm **PolyRefine** corresponding to case (ii) of Theorem 3 in Figure 7. Let P_j^+, P_j^- be two projection functions, such that for $s = (d_1, \ldots, d_m)$, $P_j^+(s) = d_j$ and

$P_j^-(s) = (d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$. Then $proj(S_{i,0}, j, a)$ denotes the *projection* set $\{P_j^-(s) | P_j^+(s) = a, s \in S_{i,0}\}$. Intuitively, the condition $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$ in the algorithm means that there exists $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \in proj(S_{i,0}, j, a)$ and $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \notin proj(S_{i,0}, j, b)$. According to the definition of $proj(S_{i,0}, j, a)$, $s_1 = (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in S_{i,0}$ and $s_2 = (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \notin S_{i,0}$, i.e., $s_2 \in S_{i,1}$. Note that s_1 and s_2 are only different at j-th component. Hence, the only way to separate s_1 and s_2 into different equivalence classes is that a and b have to be in different equivalence classes of $\equiv'_j$, i.e., $a \not\equiv'_j b$.

Lemma 2. *When $S_{i,x} = \emptyset$, the relation $\equiv'_j$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv_j$ and separates $S_{i,0}$ and $S_{i,1}$. Furthermore, the equivalence relation $\equiv'_j$ is the coarsest refinement of $\equiv_j$.*

Note that in symbolic presentation, the projection operation $proj(S_{i,0}, j, a)$ amounts to computing a generalized cofactor, which can be easily done by standard BDD methods. Given a function $f : D \to \{0, 1\}$, a generalized cofactor of f with respect to $g = (\bigwedge_{k=p}^{q} x_k = d_k)$ is the function $f_g = f(x_1, \ldots, x_{p-1}, d_p, \ldots, d_q, x_{q+1}, \ldots, x_n)$. In other words, f_g is the projection of f with respect to g. Symbolically, the set $S_{i,0}$ is represented by a function $f_{S_{i,0}} : D \to \{0, 1\}$, and therefore, the projection $proj(S_{i,0}, j, a)$ of $S_{i,0}$ to value a of the jth component corresponds to a cofactor of $f_{S_{i,0}}$.

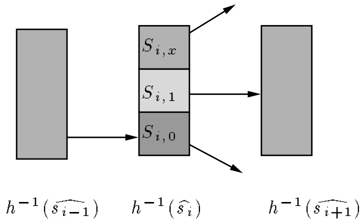

Fig. 6. Three sets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$

Algorithm PolyRefine
for j := 1 **to** m {
  $\equiv'_j := \equiv_j$
  for every $a, b \in E_j$ {
    if $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$
      then $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$ }}

Fig. 7. The algorithm **PolyRefine**

In our implementation, we use the following heuristics: We merge the states in $S_{i,x}$ into $S_{i,1}$, and use the algorithm **Polyrefine** to find the coarsest refinement that separates the sets $S_{i,0}$ and $S_{i,1} \cup S_{i,x}$. The equivalence relation computed by **PolyRefine** in this manner is not optimal, but it is a correct refinement which separates $S_{i,0}$ and $S_{i,1}$, and eliminates the spurious counterexample. This heuristic has given good results in our practical experiments.

Since according to Theorem 2, the algorithm **SplitLOOP** for loop counterexamples works analogously as **SplitPATH**, the refinement procedure for spurious loop counterexamples works analogously, too. Details are omitted due to space restrictions. Our refinement procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the $\mathrm{ACTL}^\star$ property is verified. The partitioning procedure is guaranteed to terminate since each equivalence class must contain at least one element. Thus, our method is complete.

Theorem 4. *Given a model M and an $\mathrm{ACTL}^\star$ specification φ whose counterexample is either path or loop, our algorithm will find a model $\widehat{M}$ such that $\widehat{M} \models \varphi \Leftrightarrow M \models \varphi$.*

5 Performance Improvements

The symbolic methods described in Section 4 can be directly implemented using BDDs. Our implementation uses additional heuristics which are outlined in this section. For details, we refer to our technical report [7].

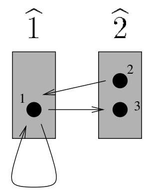

Fig. 8. A spurious loop counterexample $\langle\widehat{1},\widehat{2}\rangle^\omega$

Two-phase Refinement Algorithms. Consider the spurious loop counterexample $\widehat{T} = \langle\widehat{1},\widehat{2}\rangle^\omega$ of Figure 8. Although $\widehat{T}$ is spurious, the concrete states involved in the example contain an infinite path $\langle 1, 1, \ldots\rangle$ which is a potential counterexample. Since we know that our method is complete, such cases could be ignored. Due to practical performance considerations, however, we came to the conclusion that the relatively small effort to detect additional counterexamples is justified as a valuable heuristic. For a general loop counterexample $\widehat{T} = \langle\widehat{s}_1, \ldots, \widehat{s}_i\rangle\langle\widehat{s}_{i+1}, \ldots, \widehat{s}_n\rangle^\omega$, we therefore proceed in two phases:
(i) We restrict the model to the state space $S_{\mathrm{local}} := (\bigcup_{1\le i\le n} h^{-1}(\widehat{s_i}))$ of the counterexample and use the standard fixpoint computation for temporal formulas (see e.g. [8]) to check the property on the Kripke structure restricted to S_{local}. If a concrete counterexample is found, then the algorithm terminates.
(ii) If no counterexample is found, we use **SplitLOOP** and **PolyRefine** to compute a refinement as described above.
This two-phase algorithm is slightly slower than the original one if we do not find a concrete counterexample; in many cases however, it can speed up the search for a concrete counterexample. An analogous two phase approach is used for finite path counterexamples.

Approximation. Despite the use of partitioned transition relations it is often infeasible to compute the total transition relation of the model M [8]. Therefore, the abstract model $\widehat{M}$ cannot be computed from M directly. In previous work [2, 10], a method which we call *early approximation* has been introduced: first, abstraction is applied to the BDD representation of each transition block and then the BDDs for the partitioned transition relation are built from the already abstracted BDDs for the transition blocks. The disadvantage of early approximation is that it *over-approximates* the abstract model

$\widehat{M}$ [9]. In our approach, a heuristic individually determines for each variable cluster VC_i, if early approximation should be applied or if the abstraction function should be applied in an exact manner. Our method has the advantage that it balances overapproximation and memory usage. Moreover, the overall method presented in our paper remains complete with this approximation.

Abstractions For Distant Variables. In addition to the methods of Section 4.1, we completely abstract variables whose distance from the specification in the *variable dependency graph* is greater than a user-defined constant. Note that the variable dependency graph is also used for this purpose in the localization reduction [2, 14, 17] in a similar way. However, the refinement process of the localization reduction [14] can only turn a completely abstracted variable into a completely unabstracted variable, while our method uses intermediate abstraction functions.

6 Experimental Results

We have implemented our methodology in NuSMV [6] which uses the CUDD package [21] for symbolic representation. We performed two sets of experiments. One set is on five benchmark designs. The other was performed on an industrial design of a multimedia processor from Fujitsu [1]. All the experiments were carried out on a 200MHz PentiumPro PC with 1GB RAM memory using Linux.

The first benchmark designs are publicly available. The PCI example is extracted from [5]. The results for these designs are listed in the table.

Design	#Var	#Prop	**NuSMV+COI**				**NuSMV+ABS**			
			#COI	Time	$\lvert TR\rvert$	$\lvert MC\rvert$	#ABS	Time	$\lvert TR\rvert$	$\lvert MC\rvert$
gigamax	10(16)	1	0	0.3	8346	1822	9	0.2	13151	816
guidance	40(55)	8	30	35	140409	30467	34-39	30	147823	10670
p-queue	12(37)	1	4	0.5	51651	1155	5	0.4	52472	1114
waterpress	6(21)	4	0-1	273	34838	129595	4	170	38715	3335
PCI bus	50(89)	10	4	2343	121803	926443	12-13	546	160129	350226

In the table, the performance for an enhanced version of NuSMV with cone of influence reduction (**NuSMV + COI**) and our implementation (**NuSMV + ABS**) are compared. #Var and #Prop are properties of the designs: #Var = $x(y)$ means that x is the number of symbolic variables, and y the number of Boolean variables in the design. #Prop is the number of verified properties. The columns #COI and #ABS contain the number of symbolic variables which have been abstracted using the cone of influence reduction (#COI), and our initial abstraction (#ABS). The column ”Time” denotes the accumulated running time to verify all #Prop properties of the design. $|TR|$ denotes the maximum number of BDD nodes used for building the transition relation. $|MC|$ denotes the maximum number of *additional* BDD nodes used during the verification of the properties. Thus, $|TR| + |MC|$ is the maximum BDD size during the total model checking process. For the larger examples, we use partitioned transition relations by setting the BDD size limit to 10000.

Although our approach in one case uses 50% more memory than the traditional cone of influence reduction to *build* the abstract transition relation, it requires one magnitude less memory during *model checking*. This is an important achievement since the model checking process is the most difficult task in verifying large designs. More significant improvement is further demonstrated by the Fujitsu IP core design.

The Fujitsu IP core design is a multimedia assist (MMA-ASIC) processor [1]. The design is a system-on-a-chip that consists of a co-processor for multimedia instructions, a graphic display controller, peripheral I/O units, and five bus bridges. The RTL implementation of MM-ASIC is described in about 61,500 lines of Verilog-HDL code. After manual abstraction by engineers from Fujitsu in [22], there still remain about 10,600 lines of code with roughly 500 registers. We translated this abstracted Verilog code into 9,500 lines of SMV code. In [22], the authors verified this design using a ”navigated” model checking algorithm in which state traversal is restricted by navigation conditions provided by the user. Therefore, their methodology is not complete, i.e., it may fail to prove the correctness even if the property is true. Moreover, the navigation conditions are usually not automatically generated.

In order to compare our model checker to others, we tried to verify this design using two state-of-the-art model checkers - Yang's SMV [23] and NuSMV [6]. We implemented the cone of influence reduction for NuSMV, but not for Yang's SMV. Both NuSMV+COI and Yang's SMV failed to verify the design. On the other hand, our system abstracted 144 symbolic variables and with three refinement steps, successfully verified the design, and found a bug which has not been discovered before.

7 Conclusion and Future Work

We have presented a novel abstraction refinement methodology for symbolic model checking. The advantages of our methodology have been demonstrated by experimental results. We believe that our technique is general enough to be adapted for other forms of abstraction. There are many interesting avenues for future research. First, we want to find efficient approximation algorithms for the NP-complete separation problem encountered during the refinement step. Moreover, in a recent paper [4], the fragment of ACTL* that admits “trace”-like counterexamples (of a potentially more complicated structure than paths and loops) has been characterized; we plan to extend our refinement algorithm to this language. Since the symbolic methods described in this paper are not tied to representation by BDDs, we will also investigate how they can be applied to recent work on symbolic model checking without BDDs [3]. We are currently applying our technique to verify other large examples.

References

1. Fujitsu aims media processor at DVD. *MicroProcessor Report*, pages 11–13, 1996.
2. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, volume 697 of *LNCS*, pages 29–40, 1993.
3. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference*, pages 317–320, 1999.
4. F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On ACTL formulas having deterministic counterexamples. Technical report, Vienna University of Technology, 1999. available at http://www.kr.tuwien.ac.at/research/reports/index.html.

5. P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying IP-core based System-On-Chip design. In *IEEE ASIC*, September 1999.
6. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. Technical Report CMU-CS-00-103, Computer Science, Carnegie Mellon University, 2000.
8. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.
9. E. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: a technique for using abstraction in model checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 172–186, 1999.
10. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.
11. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 160–171. Springer Verlag, July 1999.
12. Shankar G. Govindaraju and David L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *LNCS*, pages 72–83, June 1997.
14. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
15. Y. Lakhnech. personal communication. 2000.
16. 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, November 1996.
17. J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 316–327. Springer Verlag, 1999.
18. A. Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems*. PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.
19. A. Pardo and G.D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.
20. H. Saidi and N. Shankar. Abstract and model checking while you prove. In *Computer-Aided Verification*, number 1633 in LNCS, pages 443–454, July 1999.
21. F. Somenzi. CUDD: CU decision diagram package. Technical report, University of Colorado at Boulder, 1997.
22. K. Takayama, T. Satoh, T. Nakata, and F. Hirose. An approach to verify a large scale system-on-chip using symbolic model checking. In *International Conference of Computer Design*, pages 308–313, 1998.
23. B. Yang et al. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, volume 1522 of *LNCS*. Springer Verlag, 1998.