Counterexample-Guided Abstraction Refinement for Symbolic Model Checking

EDMUND CLARKE

Carnegie Mellon University, Pittsburgh, Pennsylvania

ORNA GRUMBERG

The Technion, Haifa, Israel

SOMESH JHA

University of Wisconsin, Madison, Wisconsin

YUAN LU

Broadcom Co., San Jose, California

AND

HELMUT VEITH

Vienna University of Technology, Wien, Austria

Abstract. The state explosion problem remains a major hurdle in applying symbolic model checking to large hardware designs. State space abstraction, having been essential for verifying designs of industrial complexity, is typically a manual process, requiring considerable creativity and insight.

A preliminary version of this article has been presented at the 2000 Conference on Computer-Aided Verification (CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *Proceedings of the 2000 Conference on Computer-Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, New York. (Full version available as Tech. Rep. CMU-CS-00-103. Carnegie Mellon University, Pittsburgh, PA.))

A related survey on the state explosion problem has appeared in volume 2000 of Springer Lecture Notes (CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2001. Progress on the state explosion problem in model checking. In *informatics, 10 Years Back, 10 Years Ahead*. Lecture Notes in Computer Science, vol. 2000. Springer-Verlag, New York, pp. 176–194).

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, the Defense Advanced Research Projects Agency (DARPA) under Air Force contract No. F33615-00-C-1701, the Max Kade Foundation and the 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.

Authors' addresses: E. Clarke, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891, e-mail: Edmund.Clarke@cs.cmu.edu; O. Grumberg, Computer Science Department, The Technion, Technion City, Haifa 32000, Israel, e-mail: orna@cs.technion.ac.il; S. Jha, Computer Sciences Department, University of Wisconsin—Madison, 1210 West Dayton Street, Madison, WI 53706, e-mail: jha@cs.wisc.edu; Y. Lu, Enterprise Switching Division, Broadcom Company, 3152 Zanker Road, San Jose, CA 95134, e-mail: ylu@broadcom.com; H. Veith, Technische Universität Wien, Institut für Informationssysteme 184/2, Abt. für Datenbanken und Artificial Intelligence, Favoritenstraße 9, A-1040 Wien, Austria, e-mail: veith@dbai.tuwien.ac.at.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2003 ACM 0004-5411/03/0900-0752 $5.00

In this article, we present an automatic iterative abstraction-refinement methodology that extends symbolic model checking. In our method, 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 that analyze such counterexamples and refine the abstract model correspondingly. We describe aSMV, a prototype 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.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*temporal logic*

General Terms: Verification

Additional Key Words and Phrases: Abstraction, temporal logic, hardware verification, symbolic model checking

1. *Introduction*

During the last two decades, temporal logic model checking [Clarke and Emerson 1981; Clarke et al. 1983] has become an important application of logic in computer science. Temporal logic model checking is a technique for verifying that a system satisfies its specification by (i) representing the system as a Kripke structure, (ii) writing the specification in a suitable temporal logic, and (iii) algorithmically checking that the Kripke structure is a model of the specification formula. Model checking has been successfully applied in hardware verification, and is emerging as an industrial standard tool for hardware design. For an extensive overview of model checking, please refer to Clarke et al. [1999a].

The main technical challenge in model checking is the *state explosion problem* that can occur if the system being verified has components that make transitions in parallel. A fundamental breakthrough was made in the fall of 1987 by Ken McMillan, who was then a graduate student at Carnegie Mellon. He argued that larger systems could be handled if transition relations were represented implicitly with ordered binary decision diagrams (BDDs) [Bryant 1986]. By using the original model checking algorithm with the new representation for transition relations, he was able to verify some examples that had more than 10^{20} states [Burch et al. 1992; McMillan 1993]. He made this observation independently of the work by Coudert et al. [1989] and [Pixley 1990; Pixley et al. 1991, 1992] on using BDDs to check equivalence of deterministic finite-state machines. Since then, various refinements of the BDD-based techniques by other researchers have pushed the state count up to more than 10^{120} [Burch et al. 1991]. The widely used symbolic model checker SMV [McMillan 1993] is based on these ideas. The results reported in this paper have been implemented in NuSMV [Cimatti et al. 1998], a state-of-the-art reimplementation of SMV.

Despite the success of symbolic methods, the state explosion problem remains a major hurdle in applying model checking to large industrial designs. A number of state reduction approaches have been proposed to reduce the number of states in the model. State reduction techniques include symmetry reductions [Clarke et al. 1996; Emerson and Sistla 1996; Emerson and Trefler 1999; Ip and Dill 1996; Jensen 1996], partial order reductions [Godefroid et al. 1996; Peled 1993], and abstraction techniques [Cousot and Cousot 1977; Graf and Saïdi 1997; Long 1993; Clarke et al. 1994]. Among these techniques, abstraction is considered the most general and flexible for handling the state explosion problem.

Intuitively, abstraction amounts to removing or simplifying details as well as removing entire components of the original design that are irrelevant to the property under consideration. The evident rationale is that verifying the simplified ("abstract") model is more efficient than verifying the original model. The information loss incurred by simplifying the model however has a price: verifying an abstract model potentially leads to wrong results, at least if performed naively. Consequently, abstraction techniques can be distinguished by how they control the information loss. *Over-approximation* [Clarke et al. 1994; Kurshan 1994] and *under-approximation* techniques [Lee et al. 1996; Pardo and Hachtel 1998] keep the error one-sided, that is, they admit only false negatives (erroneous counterexamples) and false positives, respectively. Over-approximation techniques systematically release constraints, thus enriching the behavior of the system. They establish a relationship between the abstract model and the original one such that correctness at the abstract level implies correctness of the original system. In contrast, under-approximation techniques systematically remove irrelevant behavior from the system so that a specification violation at the abstract level implies a specification violation of the original system. Other techniques based on abstract interpretation [Cousot and Cousot 1977; Dams et al. 1997a] or 3-valued logics [McMillan 1999a; Bruns and Godefroid 1999; Sagiv et al. 1999] typically produce both correct positives and correct negatives, but may terminate with an unknown or approximate result. We refer the reader to Section 4.1 for an overview of additional related work.

In practice, abstraction based methods have been essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, which requires considerable creativity and insight. In order for model checking to be used more widely in industry, *automatic* techniques are needed for generating abstractions.

This article describes a new *counterexample-guided abstraction technique* that extends the general framework of *existential abstraction* [Clarke et al. 1994]. The new methodology integrates symbolic model checking and existential abstraction into a unified framework. The method is efficient, fully automatic and based on symbolic algorithms. Starting with a relatively small skeletal representation of the system to be verified, we show how to compute increasingly precise abstract representations of the system. The key step is to extract information from false negatives due to over-approximation. Such artificial specification violations are witnessed by what we call "spurious counterexamples". Our method is complete for an important fragment of ACTL*, precisely for the fragment of ACTL* that admits counterexamples in the form of finite or infinite traces (i.e., finite traces followed by loops).

In this article, we aim to give a complete picture of the counterexample-guided refinement method, ranging from the theoretical foundations of the method to implementation-related issues and practical experiments involving a large Fujitsu IP core design.

1.1. TECHNICAL APPROACH. Throughout the article, we work in the framework of existential abstraction. Existential abstraction computes an over-approximation of the original model that *simulates* the original model. Thus, when a specification in the temporal logic ACTL* 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 that is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior that caused the erroneous counterexample is eliminated. This gives rise to a sequence of increasingly precise abstract models till the specification is either proved or disproved by a counterexample.

This article shows how this general counterexample-guided abstraction framework can be applied effectively to branching-time-symbolic model checking. The following list emphasizes the main technical contributions of the article:

(1) We describe a method to extract abstraction functions from the program text; in this method, the control flow predicates of the program are used to factor the state space into equivalence classes that are acting as abstract states. While reminiscent of predicate abstraction [Graf and Saïdi 1997], this method fits into the existential abstraction framework proposed in Clarke et al. [1999b, 1994], and is used to obtain the initial abstract model.

(2) For performance reasons, abstraction functions in the literature are often defined only for individual variables, and the abstraction function is obtained as a composition of individual abstraction functions. In contrast to this, we use abstraction functions that are defined on finite tuples of variables called *variable clusters* that are extracted from the program code. This approach allows one to express conditions involving several variables (such as $x < y$) in the abstraction function, and thus helps to keep the size of the abstract state space small.

(3) We introduce symbolic algorithms to assert whether abstract counterexamples are spurious, or have corresponding concrete counterexamples. If a counterexample is spurious, 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 (the "failure state") needs to be split into less abstract states by refining the equivalence classes in such a way that the spurious counterexample is eliminated. Thus, a more refined abstraction function is obtained.

(4) Note that there may be many ways of splitting the failure 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 symbolic polynomial-time refinement algorithm that gives a suboptimal but sufficiently good refinement of the abstraction function. The applicability of our heuristic algorithm is confirmed by our experiments.

Summarizing, our technique has a number of advantages:

—The technique is complete for an important fragment of the temporal logic ACTL*.

—The initial abstraction and the refinement steps are efficient and entirely automatic and all algorithms are symbolic.

—In comparison to methods like the localization reduction [Kurshan 1994], we distinguish more degrees of abstraction for each variable. Thus, the changes in the refinement are potentially finer in our approach. The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small.

1.2. PRACTICAL EXPERIMENTS. We have implemented a prototype tool aSMV based on NuSMV [Cimatti et al. 1998] and applied it to a number of benchmark designs. In addition we have used it to debug a large IP core being developed at Fujitsu [1996]. The design has about 350 symbolic variables that correspond to about 500 latches. Before using our methodology, we implemented the *cone of influence* [Clarke et al. 1999a, Page 193] reduction 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 [Yang et al. 1998] 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. aSMV automatically abstracted 144 symbolic variables and verified the design using just three refinement steps.

1.3. RELATED WORK. Our method extends the existential abstraction framework developed by Clarke et al. [1994]. This framework will be outlined in Section 2. In our methodology, *atomic formulas* that describes the model are automatically extracted from the program. The atomic formulas are similar to the *predicates* used for abstraction by Graf and Saïdi [1997]. 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. A more detailed comparison to predicate abstraction will be provided in Section 4.2.2.

Using counterexamples to refine abstract models has been investigated by a number of other researchers beginning with the *localization reduction* of Kurshan [1994]. He models a concurrent system as a composition of L-processes $L_1, \ldots, L_n$ (L-processes are described in detail in Kurshan [1994]). 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 Balarin and Sangiovanni-Vincentelli [1993]. 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-Nielsen and Andersen [1999]. Their model checker uses upper and lower approximations in order to handle entire CTL. Their approximation techniques enable them to avoid rechecking the entire model after each refinement step while guaranteeing completeness. As in Balarin and Sangiovanni-Vincentelli [1993] and Kurshan [1994], 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 [Lee et al. 1996; Pardo 1997; Pardo and Hachtel 1998] 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 our article in order to achieve better performance. A recent technique proposed by Govindaraju and Dill [1998] 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. Another counterexample-guided refinement approach supported by good experimental results has been presented by Govindaraju and Dill [2000]. The ideas presented in this paper are similar at a high level, but their work is restricted to safety properties and uses overlapping projections as underlying approximation scheme. A hamming distance heuristic is used to obtain the registers of the design that are to be refined. In comparison to our work, we put specific emphasis on the initial abstraction by predicate abstraction, and allow for a more general class of specifications.

Many abstraction techniques can be viewed as applications of the abstract interpretation framework [Cousot and Cousot 1977, 1999; Sifakis 1983]. Given an abstract domain, abstract interpretation provides a general framework for automatically "interpreting" systems on an abstract domain. The classical abstract interpretation framework was used to prove safety properties, and does not consider temporal logic or model checking. Bjorner et al. [1997] use abstract interpretation to automatically generate invariants for infinite-state systems. Abstraction techniques for various fragments of CTL* have been discussed in Dams et al. [1993, 1997a]. These abstraction techniques have been extended to the μ-calculus [Dams et al. 1997b; Loiseaux et al. 1995].

Abstraction techniques for *infinite state systems* are crucial for successful verification [Abdulla et al. 1999; Bensalem et al. 1992; Lesens and Saïdi 1997; Manna et al. 1998]. In fact, Graf and Saïdi [1997] have proposed the above-mentioned *predicate abstraction* techniques to abstract an infinite state system into a finite state system. Later, a number of optimization techniques have been developed in Bensalem et al. [1998], Das et al. 1999; Das and Dill 2001. Saïdi and Shankar

[1999] have integrated predicate abstraction into the PVS system, which could easily determine when to abstract and when to model check. Variants of predicate abstraction have been used in the Bandera Project [Dwyer et al. 2001] and the SLAM project [Ball et al. 2001].

Colón and Uribe [1998] have presented a way to generate finite-state abstractions using a decision procedure. Similar to predicate abstraction, their abstraction is generated using abstract Boolean variables. One difference between our approach and the predicate abstraction based approaches is that the latter tries to build an abstract model on-the-fly while traversing the reachable state sets. Our approach tries to build the abstract transition relation directly.

Wolper and Lovinfosse [1989] have verified *data independent* systems using model checking. In a data independent system, the data values never affect the control flow of the computation. Therefore, the datapath can be abstracted away entirely. Van Aelten et al. [1992] have discussed a method for simplifying the verification of synchronous processors by abstracting away the data path. Abstracting away the datapath using uninterpreted function symbols is very useful for verifying pipeline systems [Berezin et al. 1998; Burch and Dill 1994; Jones et al. 1998]. A number of researchers have modeled or verified industrial hardware systems using abstraction techniques [Fura et al. 1993; Graf 1994; Ho et al. 1998; Hojati and Brayton 1995]. In many cases, their abstractions are generated manually and combined with theorem proving techniques [Rushby 1999; Rusu and Singerman 1999]. Dingel and Filkorn [1995] have used data abstraction and assume-guarantee reasoning combined with theorem proving techniques to verify infinite state systems. Recently, McMillan [1996b] has incorporated a new type of data abstraction, assume-guarantee reasoning and theorem proving techniques in his Cadence SMV system.

1.4. ORGANIZATION OF THE ARTICLE. This article is organized as follows: Section 2 provides definitions, terminology and a formal background on model checking. Section 3 reviews and adapts the existential abstraction framework. The core of the article is Section 4, which describes the new approach and the algorithms put forward in this article. The proofs of the complexity results are postponed to Section 5. This is followed by Sections 6 and 7 which concentrate on practical performance improvements, and experiments. Future research directions are discussed in Section 8.

2. *Fundamentals of Model Checking*

2.1. KRIPKE STRUCTURES AND TEMPORAL LOGIC. In model checking, the system to be verified is formally represented by a finite *Kripke structure*, a directed graph whose vertices are labeled by sets of atomic propositions. Vertices and edges are called *states* and *transitions* respectively. One or more states are considered to be *initial states*. Thus, a Kripke structure over a set of atomic propositions A is a tuple $K = (S, R, L, I)$ where S is the set of states, $R \subseteq S^2$ is the set of transitions, $I \subseteq S$ is the nonempty set of initial states, and $L : S \to 2^A$ labels each state by a set of atomic propositions A. A *path* is an infinite sequence of states, $\pi = \langle s_0, s_1, \dots \rangle$ such that for $i \geq 0$, $(s_i, s_{i+1}) \in R$. Given a path π, π^i denotes the infinite path $\langle s^i, s^{i+1}, \dots \rangle$ we assume that the transition relation R is *total*, that is, all states have positive out-degree. Therefore, each finite path can be extended into an infinite path.

CTL* is an extension of propositional logic obtained by adding *path quantifiers* and *temporal operators*.

Path quantifiers:		**Temporal Operators**:	
A	"for every path"	$\mathbf{X}p$	"p holds next time"
E	"there exists a path"	$\mathbf{F}p$	"p holds sometime in the future"
		$\mathbf{G}p$	"p holds globally in the future"
		$p\,\mathbf{U}q$	"p holds until q holds"
		$p\,\mathbf{R}\,q$	"q is released when p becomes false"

There are two types of formulas in CTL*: *state formulas* (which are evaluated on states) and *path formulas* (which evaluated on paths). The syntax of both state and path formulas is given by the following rules:

—If $p \in$ A, then p is a state formula.

—If f and g are state formulas, then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.

—If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

—If f is a state formula, then f is also a path formula.

—If f and g are state formulas, then $\mathbf{X}\ f$, $\mathbf{F}\ \ f$, $\mathbf{G}\ \ f$, $f\ \mathbf{U}\ \ g$ and $f\ \mathbf{R}\ \ g$ are path formulas.

The semantics for CTL* is given in Appendix A.

ACTL* is the fragment of CTL* where only the path operator **A** is used, and negation is restricted to atomic formulas. An important feature of ACTL* is the existence of counterexamples. For example, the specification $\mathbf{AF}p$ denotes "*on all paths, p holds sometime in the future.*" If the specification $\mathbf{AF}p$ is violated, then there exists an infinite path where p never holds. This path is called a counterexample of $\mathbf{AF}p$. In this article, we focus on counterexamples which are finite or infinite paths.

2.2. THE MODEL CHECKING PROBLEM. Given a Kripke structure $M = (S, R, I, L)$ and a specification φ in a temporal logic such as CTL*, the *model checking problem* is the problem of finding all states s such that $M, s \models \varphi$ and checking if the initial states are among these. An explicit state model checker is a program which performs model checking directly on a Kripke structure.

THEOREM 2.1 [CLARKE ET AL. 1983; CLARKE JR. ET AL. 1986]. *Explicit state CTL model checking has time complexity* $O(|M||\varphi|)$.

CTL is a subfragment of CTL*, where each path quantifier **A** or **E** is immediately followed by a temporal operator, for example, $\mathbf{AF}p$ is a CTL formula, but $\mathbf{AFG}p$ is not. Model checking algorithms are usually fixed point algorithms that exploit the fact that temporal formulas can be expressed by fixed point formulas. In practice, systems are described by programs in finite state languages such as SMV or VERILOG. These programs are then compiled into equivalent Kripke structures.

Example 2.2. In the verification system SMV, the state space S of a Kripke structure is given by the possible assignments to the system variables. Thus, a system with 3 variables x, y, *reset* and variable domains $D_x = D_y = \{0, 1, 2, 3\}$ and $D_{reset} = \{0, 1\}$ has state space $S = D_x \times D_y \times D_{reset}$, and $|S| = 32$.

The binary transition relation R is defined by transition blocks which for each variable define possible values in the next time cycle, as in the following example:

```
init(reset) := 0;          init(x) := 0;        init(y) := 1;
next(reset) := {0, 1};     next(x) := case      next(y) := case
                               reset = 1 : 0;       reset = 1 : 0;
                               x < y : x + 1;       (x = y) ∧ ¬(y = 2) : y + 1;
                               x = y : 0;           (x = y) : 0;
                               else : x;            else : y;
                           esac;                esac;
```

Here, **next**(*reset*) := {0, 1} means that the value of *reset* is chosen nondeterministically. Such situations occur frequently when *reset* is controlled by the environment, or when the model of the system is too abstract to determine the values of *reset*. For details about the SMV input language, we refer the reader to McMillan [1993].

The main practical problem in model checking is the so-called *state explosion problem* caused by the fact that the Kripke structure represents the *state space* of the system under investigation, and thus its size is potentially *exponential* in the size of the system description. Therefore, even for systems of relatively modest size, it is often impossible to compute their Kripke structures explicitly. In the rest of this article, we will focus on a combination of two techniques, *symbolic model checking* and *abstraction* which alleviate the state explosion problem.

2.3. Symbolic Model Checking.

In *symbolic model checking*, the transition relation of the Kripke structure is not explicitly constructed, but instead a Boolean function is computed which represents the transition relation. Similarly, sets of states are also represented by Boolean functions. Then, fixed point characterizations of temporal operators are applied to the Boolean functions rather than to the Kripke structure. Since in many practical situations the space requirements for Boolean functions are exponentially smaller than for explicit representation, symbolic verification is able to alleviate the state explosion problem in these situations. These Boolean functions are represented as *ordered binary decision diagrams* (*BDDs*). A short description on BDDs can be found in Section A.

A symbolic model checking algorithm is an algorithm whose variables denote not single states, but *sets of states* that are represented by Boolean functions (usually as BDDs, see Clarke et al. [1999a, Chap. 5]). Therefore, symbolic algorithms use only such operations on sets which can be translated into BDD operations. For example, union and intersection of sets correspond to disjunction and conjunction respectively. Binary Decision Diagrams have been a particularly useful data structure for representing Boolean functions; despite their relative succinctness they provide canonical representations of Boolean functions, and therefore expressions of the form $S_1 = S_2$, which are important in fixed point computations, can be evaluated very efficiently.

Image computation is the task of computing for a given set S of states the set of states

$$Img(S, R) := \{t \mid \exists s. R(s, t) \wedge s \in S\}.$$

Since all temporal operators can be expressed as a combination of fixed point computations and image computations, image computation is central to verification tasks. In contrast to the other operations, however, image computation is not a simple BDD operation, and therefore presents of the major bottlenecks in verification. Part of the reason for this, ironically, is the fact that it is in general *not feasible* to construct a single BDD for the transition relation R. Instead, R is represented as the conjunction of several BDDs [McMillan 1996]. In Section 3.1, we shall address the problem of computing an abstraction of R without actually computing R.

Practical experiments show that the performance of symbolic methods is highly unpredictable. This phenomenon can be partially explained by complexity theoretic results which state that BDD representation does not improve worst case complexity. In fact, it has been shown [Feigenbaum et al. 1999; Veith 1998a] that representing a decision problem in terms of exponentially smaller BDDs usually increases its worst case complexity exponentially. For example, the problem of deciding $\mathbf{EF}p$ (reachability) is complete for nondeterministic logspace-NL, while in BDD representation it becomes complete for PSPACE. Similar results can be shown for other Boolean formalisms and are closely tied to principal questions in structural complexity theory [Gottlob et al. 1999; Veith 1997, 1998b].

2.4. SIMULATION AND PRESERVATION OF ACTL*. Given two Kripke structures M and M' with A $\supseteq A'$, a relation $H \subseteq S \times S'$ is a *simulation relation* between M and M' if and only if for all $(s, s') \in H$, the following conditions hold.

(1) $L(s) \cap A' = L'(s')$.

(2) For each state s_1 such that $(s, s_1) \in R$, there exists a state s'_1 with the property that $(s', s'_1) \in R'$ and $(s_1, s'_1) \in H$.

We say that M' *simulates* M (denoted by $M \preceq M'$) if there exists a simulation relation H such that for every initial state s_0 in M there exists an initial state s'_0 in M' for which $(s_0, s'_0) \in H$. It is easy to see that the simulation relation $\preceq$ is a preorder on the set of Kripke structures. The following classic result is one of the cornerstones of the abstraction refinement framework [Clarke et al. 1994a, 1999].

THEOREM 2.3. *For every ACTL* formula φ over atomic propositions A', if $M \preceq M'$ and $M' \models \varphi$, then $M \models \varphi$.*

3. *Abstraction*

Intuitively speaking, existential abstraction amounts to partitioning the states of a Kripke structure into clusters, and treating the clusters as new abstract states (see Figure 1).

Formally, an abstraction function h is described by a surjection $h : S \to \widehat{S}$ where $\widehat{S}$ is the set of *abstract states*. The surjection h induces an equivalence relation $\equiv_h$ on the domain S in the following manner: let d, e be states in S, then

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

Since an abstraction can be represented either by a surjection h or by an equivalence relation $\equiv_h$, we sometimes switch between these representations. When the context is clear, we often write $\equiv$ instead of $\equiv_h$.

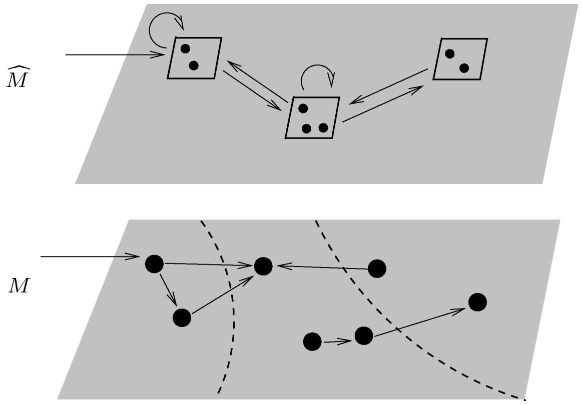


FIG. 1. Existential Abstraction. M is the original Kripke structure, and $\widehat{M}$ the abstracted one. The dotted lines in M indicate how the states of M are clustered into abstract states.

The *abstract Kripke structure* $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L})$ generated by the abstraction function h is defined as follows:

(1) $\widehat{I}(\widehat{d})$ iff $\exists d(h(d) = \widehat{d} \wedge I(d))$.
(2) $\widehat{R}(\widehat{d_1}, \widehat{d_2})$ iff $\exists d_1 \exists d_2 (h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$.
(3) $\widehat{L}(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$.

Sometimes we write $\widehat{M} = \widehat{M}_h = (\widehat{S}_h, \widehat{I}_h, \widehat{R}_h, \widehat{L}_h)$ to emphasize that $\widehat{M}$ is generated by h.

Next we introduce a condition for abstraction functions which guarantees that for a given specification, no false positives are possible on the abstract model. (Formally, this will be stated in Theorem 3.3 below.)

An abstraction function h is *appropriate* for a specification φ if for all atomic subformulas f of φ and for all states d and e in the domain S such that $d \equiv_h e$ it holds that $d \models f \Leftrightarrow e \models f$. If φ is understood from the context, we just say that h is appropriate. h is appropriate for a set $\mathcal{F}$ of formulas if h is appropriate for all $f \in \mathcal{F}$. 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 $\widehat{L}(\widehat{d})$, that is, collapsing a set of concrete states into an abstract state does not lead to contradictory labels. The following proposition is an immediate consequence of the definitions:

PROPOSITION 3.1. *If h is appropriate for φ, then (i) all concrete states in an equivalence class of $\equiv_h$ share the same labels; (ii) the abstract states inherit all labels from each state in the respective equivalence classes; (iii) the labels of the abstract states are consistent.*

In other words, $d \equiv_h d'$ implies $L(d) = L(d')$, $h(d) = \widehat{d}$ implies $\widehat{L}_h(\widehat{d}) = L(d)$, and $\widehat{L}_h(d)$ is consistent.

It is easy to see that $\widehat{M}$ contains less information than M. Thus, model checking the structure $\widehat{M}$ potentially leads to incorrect results. The following theorem shows that at least for ACTL*, specifications which are correct for $\widehat{M}$ are correct for M as well.

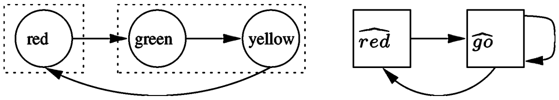


FIG. 2. Abstraction of a US traffic light.

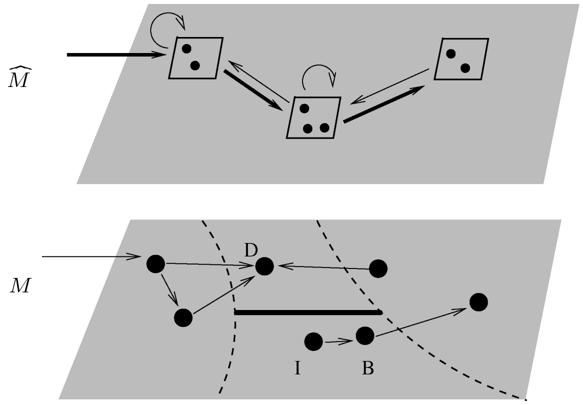


FIG. 3. The abstract path in $\widehat{M}$ (indicated by the thick arrows) is spurious. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line in M.

LEMMA 3.2. *Let h be appropriate for the* ACTL* *specification φ and M be defined over the atomic propositions in φ. Then $M \preceq \widehat{M}_h$.*

PROOF. Choose the simulation relation H to be $\{(s, h(s)) \mid s \in S\}$. Then the definition is trivially satisfied. □

We conclude that in the existential abstraction framework presented here, there are no false positives:

THEOREM 3.3. *Let h be appropriate for the* ACTL* *specification φ. Then $\widehat{M}_h \models \varphi$ implies $M \models \varphi$.*

PROOF. Immediately from Theorem 2.3 and Lemma 3.2 □

On the other hand, the following example shows that if the abstract model invalidates an ACTL* specification, *the actual model may still satisfy the specification*.

Example 3.4. Assume that for a US traffic light controller (see Figure 2), we want to prove $\psi = \mathbf{AG}\,\mathbf{AF}(state = red)$ using the abstraction function $h(red) = \widehat{red}$ and $h(green) = h(yellow) = \widehat{go}$. It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite abstract trace $\langle \widehat{red}, \widehat{go}, \widehat{go}, \dots \rangle$ that invalidates the specification.

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

Let us consider the situation outlined in Figure 3. We see that the abstract path does not have a corresponding concrete path. Whichever concrete path we follow, we will end up in state D, from which we cannot go any further. Therefore, D is called a *deadend state*. On the other hand, the *bad state* is state B, because it made us believe that there is an outgoing transition. In addition, there is the set of irrelevant states I that are neither deadend nor bad, and it is immaterial whether states in I are combined with D or B. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line.

3.1. APPROXIMATION FOR EXISTENTIAL ABSTRACTION. As argued above, it is usually computationally too expensive to compute existential abstraction directly [Long 1993]. Instead of building $\widehat{M}_h$ directly, approximation is often used to reduce the complexity. If a Kripke structure $\tilde{M} = (S_h, \tilde{I}, \tilde{R}, \widehat{L}_h)$ satisfies

(1) $\widehat{I}_h \subseteq \tilde{I}$ and
(2) $\widehat{R}_h \subseteq \tilde{R}$,

then we say that $\tilde{M}$ *approximates* $\widehat{M}_h$. Intuitively, if $\tilde{M}$ approximates $\widehat{M}_h$, then $\tilde{M}$ is *more* abstract than $\widehat{M}_h$, that is, $\tilde{M}$ has more behaviors than $\widehat{M}_h$. (In the terminology of Section 1, $\tilde{M}$ is an over-approximation.)

THEOREM 3.5. *Let h be appropriate for the* ACTL* *specification φ. Then $\tilde{M}$ simulates $\widehat{M}_h$, that is, $\widehat{M}_h \preceq \tilde{M}$.*

PROOF. A simulation relation is given by the identity mapping that maps each state in M to the same state in $\widehat{M}_h$ – note that the sets of states in M and $\widehat{M}_h$ coincide. □

Remark 3.6. As the proof shows, approximation is much simpler than existential abstraction, because it does not affect the set of states. By approximating the set of transitions from above, we trade complexity against more spurious behavior.

Since $\preceq$ is a preorder, $M \preceq \widehat{M}_h \preceq \tilde{M}$ in accordance with Theorem 3.3 and Theorem 3.5. Clarke et al. [1994] define a practical transformation $\mathcal{T}$ called *early approximation* that applies the existential abstraction operation directly to variables at the innermost level of the formula. This transformation generates a new structure $\tilde{M}_{\mathcal{T}}$ as follows. Assume that $R = R_1 \wedge \cdots \wedge R_n$ where each R_i defines the transition relation for a single variable. Then, we apply abstraction to each R_i separately, that is,

$$\mathcal{T}(R) = (R_1)_h \wedge \cdots \wedge (R_n)_h$$

and analogously for I. Finally, $\tilde{M}_{\mathcal{T}}$ is given by $(S_h, \mathcal{T}(I), \mathcal{T}(R), \widehat{L}_h)$. As a simple example, consider a system M, which is a synchronous composition of two systems M_1 and M_2, or in other words $M = M_1 \| M_2$. Both M_1 and M_2 define the transition of one variable. In this case, $\tilde{M}_{\mathcal{T}}$ is equal to $\widehat{(M_1)}_h \| \widehat{(M_2)}_h$. Note that the existential abstraction operation is applied to each process individually. Since $\mathcal{T}$ is applied at the innermost level, abstraction can be performed before building the BDDs for the transition relation. This abstraction technique is usually fast and easy to implement. However, it has potential limitations in checking certain properties. Since $\tilde{M}_{\mathcal{T}}$ is a coarse abstraction, there exist many properties that cannot be checked on $\tilde{M}_{\mathcal{T}}$ but can still be verified using a finer approximation. The following small example will highlight some of these problems.

Example 3.7. Consider a Kripke structure for a simple traffic light example, where variable t describes the three possible values $D_t = \{r, g, y\}$ (red, green, yellow) of the traffic light, and variable c describes the two possible values

$D_c = \{s, d\}$ (stop, drive) of the car. The transition relation is described by the following two transition blocks:

```
init($t$) := $r$;
next($t$) := case
    $t = r : \{r, g\}$;
    $t = g : \{g, y\}$;
    $t = y : \{y, r\}$;
esac;
```

```
init($c$) := $s$;
next($c$) := case
    $t = g : \{s, d\}$;
    $t \neq g : s$;
esac;
```

The traffic light example is illustrated in Figure 4. First, two Kripke structures M_t and M_c are constructed from the transition blocks.[1] The Kripke structure for the system is obtained by composing M_t and M_c, yielding the Kripke structure $M_t \parallel M_c$.

The safety property we want to prove is that when the traffic light is red, the automobile should not be driving. This can be written in ACTL as follows:

$$\varphi \equiv \mathbf{AG}[\neg(t = r \wedge c = d)]$$

It is easy to see that the property φ is true over $M_t \parallel M_c$.

Let us consider an abstraction function abs which maps the values g and d to w (such as in "walk"). If we apply this abstraction *after* composing M_t and M_c, the state (r, d) remains unreachable, and the property φ still holds. If however, we use the transformation $\mathcal{T}$, which applies abstraction *before* we compose M_t and M_c, property φ does not hold as (r, d) is reachable. We see that when we first abstract the individual components and then compose we may introduce too many spurious behaviors.

It is desirable to obtain an approximation structure $\tilde{M}$ which is more precise than the structure $\tilde{M}_{\mathcal{T}}$ obtained by the technique proposed in Clarke et al. [1994]. All the transitions in the abstract structure $\widehat{M}_h$ are included in both $\tilde{M}$ and $\tilde{M}_{\mathcal{T}}$. Note that the state sets of $\widehat{M}_h$, $\tilde{M}$ and $\tilde{M}_{\mathcal{T}}$ are the same and $M \preceq \widehat{M}_h \preceq \tilde{M}_{\mathcal{T}}$. In summary, $\widehat{M}_h$ is intended to be built but is computationally expensive. $\tilde{M}_{\mathcal{T}}$ is easy to build but extra behaviors are added into the structure. Our aim is to build a model $\tilde{M}$ which is computationally easier but a more refined approximation of $\widehat{M}_h$ than $\tilde{M}_{\mathcal{T}}$, that is,

$$M \preceq \widehat{M}_h \preceq \tilde{M} \preceq \tilde{M}_{\mathcal{T}}.$$

4. *Counterexample-Guided Abstraction Refinement*

4.1. OVERVIEW. For a program P and an ACTL* 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

[1] Note that in the figure a transition of M_c is labelled by a condition (**g**) referring to the status of M_t required to perform the transition on M_c, that is, green light is required for driving. This notation reflects the fact that the transition block for M_c refers to the status variable of M_t. When M_t and M_c are composed, no more synchronization between the transition blocks is necessary, and the labels vanish.

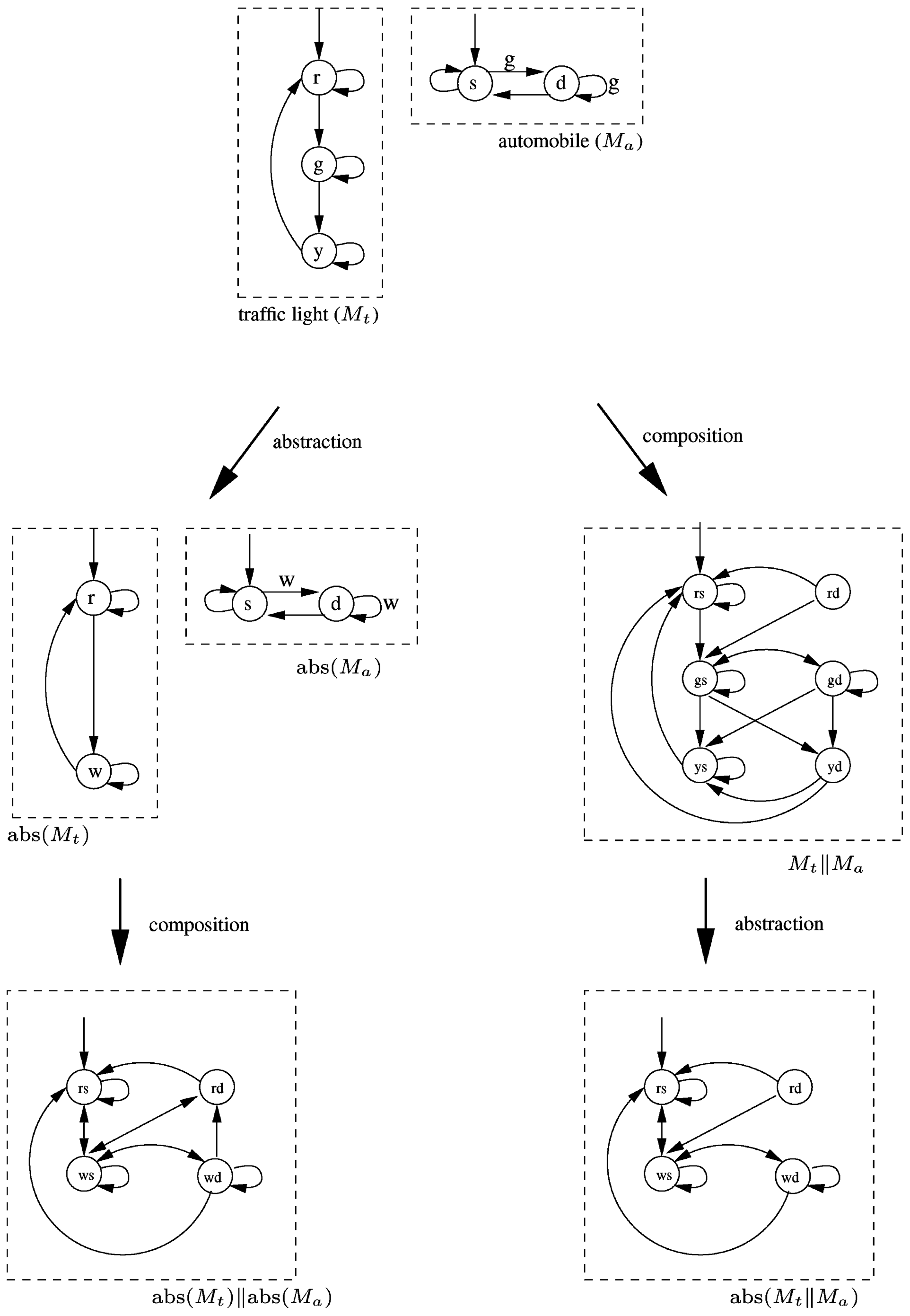


FIG. 4. Traffic light example.

```
init($v_i$) := $I_i$;
next($v_i$) := case
        $C_i^1$ : $A_i^1$;
        $C_i^2$ : $A_i^2$;
        $\cdots$ : $\cdots$;
        $C_i^k$ : $A_i^k$;
esac;
```

FIG. 5. A generic transition block for a concurrent program.

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 4.2.2.

(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 3.3). Suppose the check reveals that there is a counterexample $\widehat{T}$. We ascertain whether $\widehat{T}$ is an actual counterexample, that is, a counterexample in the unabstracted structure M. Since the model is finite, this can be achieved by "simulating" the counterexample on the actual model. 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 4.4. After refining the abstraction function, we return to step (2).

4.2. GENERATING THE INITIAL ABSTRACTION

4.2.1. *Concurrent Programs.* We describe a simple syntactic framework to formalize guarded concurrent programs. Common hardware description languages like Verilog and VHDL can easily be compiled into this language. A *program* P has a finite set of variables $V = \{v_1, \ldots, v_n\}$, where each variable v_i has an associated finite domain D_{v_i}. The set of all possible states for program P is $D_{v_1} \times \cdots \times D_{v_n}$, which we denote by D. *Expressions* are built from variables in V, constants in D_{v_i}, and function symbols in the usual way, for example, $v_1 + 3$. *Atomic formulas* are constructed from expressions and relation symbols, for example, $v_1 + 3 < 5$. Similarly, *predicates* are composed of atomic formulas using negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). Given a predicate p, 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, as shown in Figure 5. Here, $I_i \subseteq D_{v_i}$ is the initial expression for the variable v_i, each control flow condition C_i^j is a predicate, and A_i^j is an expression (see Example 2.2). The semantics of the transition block is similar to the semantics of the **case** statement in the modeling language of SMV, that is, 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.

With each program P we associate an ACTL* specification φ whose atomic formulas are constructed from expressions and relation symbols as above. For each transition block B_i let $\text{Atoms}(B_i)$ be the set of atomic formulas that appear in the conditions. Let $\text{Atoms}(\varphi)$ be the set of atomic formulas appearing in the specification φ. $\text{Atoms}(P)$ 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 $L\colon S \to 2^{\text{Atoms}(P)}$ is a labelling given by $L(d) = \{f \in \text{Atoms}(P) \mid d \models f\}$. Translating a program into a Kripke structure is straightforward and will not be described here (see Clarke et al. [1999a]) We use both D and S to refer to the set of states depending on the context.

4.2.2. *Initial Abstraction Function.* The rationale behind the abstraction function we shall define is that we identify two states if they can be distinguished neither by

(a) atomic subformulas of the specifications, nor by
(b) control flow conditions in the program.

Intuitively, clause (a) will guarantee that the abstraction function is appropriate for the specification, and clause (b) guarantees that the abstraction function does not destroy the branching structure of the program. Thus, the initial abstraction represents a “control flow skeleton” of the original system.

Formally, we define the initial abstraction function as a special case of a slightly more general definition.

Definition 4.1. Let $\mathcal{F}$ be a set of formulas. Given two states $d, e \in D$, d and e are *$\mathcal{F}$-equivalent* ($d \equiv_{\mathcal{F}} e$) if e and d cannot be distinguished by the formulas in $\mathcal{F}$, that is, for all $f \in \mathcal{F}$ we have $d \models f$ iff $e \models f$. The corresponding abstraction function is called $h_{\mathcal{F}}$.

It is easy to give a general criterion for appropriate abstraction functions:

PROPOSITION 4.2. *Let φ be an* ACTL* *specification, and $\mathcal{F} \supseteq \text{Atoms}(\varphi)$ be a set of formulas. Then $h_{\mathcal{F}}$ is appropriate for φ.*

PROOF. Suppose that $d \equiv_{\mathcal{F}} e$, but $d \models f$ and $e \not\models f$, where $f \in \text{Atoms}(\varphi)$. Since $\text{Atoms}(\varphi) \subseteq \mathcal{F}$, this contradicts the definition of $\equiv_{\mathcal{F}}$. □

The initial abstraction function is constructed in accordance with the intuition explained above:

Definition 4.3. Let P be a program and φ be an ACTL* specification. Then the *initial abstraction function* init is given by $h_{\text{Atoms}(P)}$.

Combining this with the definition given in Section 3, we obtain an initial abstraction function init satisfying

$$d \equiv_{\text{init}} e \ \text{ iff } \ h_{\text{Atoms}(P)}(d) = h_{\text{Atoms}(P)}(e) \ \text{ iff } \bigwedge_{f \in \text{Atoms}(P)} d \models f \Leftrightarrow e \models f.$$

COROLLARY 4.4. *The abstraction function* init *is appropriate for* φ.

Remark 4.5

(i) It follows from Proposition 4.2 that defining init $:= h_{\text{Atoms}(P)}$ is a *design decision* rather than a technical necessity. If the abstract space obtained in this way is too large, it is possible to replace $h_{\text{Atoms}(P)}$ by some $h_{\mathcal{X}}$ where $\mathcal{X}$ is a different superset of Atoms(φ). In particular, $\mathcal{X}$ may be derived in accordance with a different strategy than the one mentioned in clause (b) above. It is even possible to use not only atomic formulas as in Atoms(P), but also more complicated conditions involving, say, first order logic. Since we work in finite and typically small domains, such an extension is feasible.

(ii) Technically, the initial abstraction function described here can be also described as a predicate abstraction similar to Graf and Saïdi [1997] in which the abstract space contains tuples of Boolean variables $(b_1, \ldots, b_N)$ where Atoms$(P) = \{f_1, \ldots, f_N\}$ and $\text{init}(d) = (b_1, \ldots, b_N)$ iff $\bigwedge_{1 \le i \le N} b_i \Leftrightarrow (d \models f_i)$. Predicate abstraction for software, however, typically uses fewer predicates because the value of each b_i needs to be determined by a decision procedure that can have very high complexity. Moreover, the symbolic algorithms presented later do not fit well into the predicate abstraction framework.

4.2.3. *Fine Structure of Abstraction Functions.* As Example 2.2 shows, the set of states S of a Kripke structure is typically obtained as the product $D_1 \times \cdots \times D_n$ of smaller domains. A simple way to define abstraction functions is 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 $\widehat{S}$ is equal to $\widehat{D_1} \times \cdots \times \widehat{D_n}$. In this case, we write $h = (h_1, \ldots, h_n)$, and say that h is a *composite abstraction function* with components $h_1, \ldots, h_n$.

The equivalence relations $\equiv_i$ corresponding to the components h_i relate to the equivalence relation $\equiv_h$ over the entire domain $S = D_1 \times \cdots \times D_n$ in the obvious manner:

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

Remark 4.6. Previous work on existential abstraction [Clarke et al. 1994] used composite abstraction functions that are defined separately for each variable domain. Thus, 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 cannot 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 $\widehat{D}$. Thus, the approach of Clarke et al. [1994] does not exhaust the full power of abstraction functions.

It is easy to see however that the initial abstraction function init defined in Section 4.2.2 may *in principle* exhaust all combinatorial possibilities. Thus, the initial abstraction function is more general than in Clarke et al. [1994]. As argued above, it cannot always be represented as a composite abstraction function, and may therefore become intractable in practice.

It is therefore important to represent the abstraction functions we use as composite abstraction functions to the extent this is possible. To this end, we define the initial abstraction function in a different way based on the syntactic structure of the program, and show then in Theorem 4.8 that this composite abstraction is equivalent to the abstraction function init defined in Section 4.2.2. In Remark 4.2.3 at the end of this section, we discuss what to do in cases where the abstraction function cannot be represented as a composite abstraction function.

Assume that we are given a program P with n variables $V = \{v_1, \ldots, v_n\}$. 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, $D = D_{VC_1} \times \cdots \times D_{VC_m}$. We define composite abstraction functions as surjections on the domains D_{VC_i}. If the variable clusters are singletons, we obtain the composite abstraction functions of Clarke et al. [1994] as special cases.

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 Atoms(P) that is the reflexive, transitive closure of the interference relation. The equivalence class of an atomic formula $f \in$ 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 variable clusters are given by the equivalence classes of $\equiv_V$. 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 a composite abstraction function h with components $h_1, \ldots, h_m$, where each h_i is defined on $D_{VC_i} = \prod_{v \in VC_i} D_v$, the domain corresponding to the variable cluster VC_i.

The component abstraction functions h_i are defined as follows:

$$h_i(d_1, \ldots, d_k) = h_i(e_1, \ldots, e_k) \ \text{ iff } \ \bigwedge_{f \in FC_i} (d_1, \ldots, d_k) \models f \Leftrightarrow (e_1, \ldots, e_k) \models f.$$

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 abstraction function h.

Example 4.7. Consider the program P with three variables $x, y \in \{0, 1, 2\}$, and *reset* $\in$ {TRUE, FALSE} shown in Example 2.2. The set of atomic formulas is Atoms$(P) = \{(reset = \text{TRUE}), (x = y), (x < y), (y = 2)\}$. There are two formula clusters, $FC_1 = \{(x = y), (x < y), (y = 2)\}$ and $FC_2 = \{(reset = \text{TRUE})\}$. 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 {TRUE, FALSE} has two equivalence classes—one containing FALSE and the other TRUE. Therefore, we define two abstraction functions h_1: $\{0,1,2\}^2 \to \{0,1,2,3,4\}$ and h_2: $\{\text{TRUE}, \text{FALSE}\} \to \{\text{TRUE}, \text{FALSE}\}$. 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, that is, $h_2(\mathit{reset}) = \mathit{reset}$.

THEOREM 4.8. *$\widehat{M}_h$ is isomorphic to $\widehat{M}_{\text{init}}$.*

PROOF. It is sufficient to show that $\equiv_h$ and $\equiv_{\text{init}}$ are equivalent. Let π_{VC_i} be the projection function which maps a tuple $d \in D$ to the subtuple corresponding to VC_i. For each $f \in FC_i$ it holds that $d \models f$ iff $\pi_{VC_i}(d) \models f$ because by definition all $f \in FC_i$ depend only on elements of VC_i. Since the formula clusters of FC_i form a partition of Atoms(P), we obtain

$$\begin{aligned}
d \equiv_{\text{init}} e \quad &\text{iff}\\
\bigwedge_{f \in \text{Atoms}(P)} d \models f \Leftrightarrow e \models f \quad &\text{iff}\\
\bigwedge_{1 \le i \le m} \bigwedge_{f \in FC_i} \pi_{VC_i}(d) \models f \Leftrightarrow \pi_{VC_i}(e) \models f \quad &\text{iff}\\
d \equiv_h e \quad &
\end{aligned}$$

by combining the respective definitions. □

Remark 4.9. Throughout this article, we will use composite abstraction functions based on variable clusters. For actual hardware designs, our experiments have shown that this approach optimally combines feasibility and expressiveness. Some comments are in order at this point:

(i) The variable clusters are an artifact of the way we chose to define Atoms(P). By virtue of Proposition 4.2, we may construct variable clusters based on supersets of Atoms(φ) different from Atoms(P).

(ii) If a variable cluster becomes too big, we may choose to remove formulas from Atoms(P) \ Atoms(φ) in order to split variable clusters. For instance, if we have a variable cluster $\{u, v, w, x\}$ obtained from control atoms $u = v, v < w, w = x$, we may choose to disregard $v < w$ in order to obtain two variable clusters $\{u, v\}, \{w, x\}$.

(iii) It is possible to represent this problem by a hypergraph whose vertices are the variables V, and whose edges are given by $\{\mathit{var}(f) \mid f \in \text{Atoms}(P)\}$, that is, by the variables occurring in the atomic formulas we consider. (In case of binary relation symbols, the hypergraph will be a graph.) In this setting, the problem of finding suitable atoms to split variable clusters amounts to finding small cuts in the hypergraph.

4.3. MODEL CHECKING THE ABSTRACT MODEL. Given an ACTL* specification φ, an abstraction function h (assume that h is appropriate for φ), and a program P with a finite set of variables $V = \{v_1, \dots, v_n\}$, 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 3.3 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 focus on counterexamples which are either *(finite) paths* or *loops*.

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

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

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: $S_i := \mathit{Img}(S_{i-1}, R) \cap h^{-1}(\widehat{s_i})$. In the definition of S_i, $\mathit{Img}(S_{i-1}, R)$ 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 4.10. *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 nonempty.*
(iii) *For all $1 \leq i \leq n$, $S_i \neq \emptyset$.*

PROOF

(i) → (ii). Assume that $\widehat{T}$ corresponds to a concrete counterexample $T = \langle s_1, \dots, s_n\rangle$. From the definition of $\widehat{T}$, $h(s_i) = \widehat{s_i}$ and $s_i \in h^{-1}(\widehat{s_i})$ for $1 \leq i \leq n$. Since T is a trace in the concrete model, it has to satisfy the transition relation and start from initial state, that is, $R(s_i, s_{i+1})$ and $s_1 \in I$. From the definition of $h^{-1}(\widehat{T})$, it follows that $T \in h^{-1}(\widehat{T})$.

(ii) → (i). Assume that $h^{-1}(\widehat{T})$ is nonempty. We pick a trace $\langle s_1, \dots, s_n\rangle$ from $h^{-1}(\widehat{T})$. Then $\langle h(s_1), \dots, h(s_n)\rangle = \widehat{T}$, and therefore $\widehat{T}$ corresponds to a concrete counterexample.

(ii) → (iii). Assume that $h^{-1}(\widehat{T})$ is not empty. Then there exists a path $\langle s_1, \dots, s_n\rangle$ where $h(s_i) = \widehat{s_i}$ and $s_1 \in I$. Therefore, we have $s_1 \in S_1$. Let us assume that

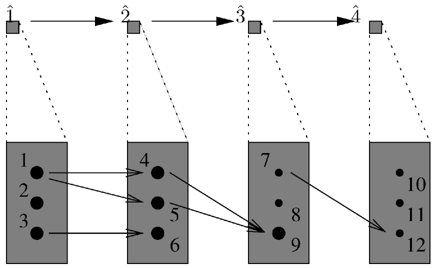


FIG. 6. An abstract counterexample.

$s_i \in S_i$. By the definition of $h^{-1}(\widehat{T})$, $s_{i+1} \in Img(s_i, R)$ and $s_{i+1} \in h^{-1}(\widehat{s_{i+1}})$. Therefore, $s_{i+1} \in Img(S_i, R) \cap h^{-1}(\widehat{s_{i+1}}) = S_{i+1}$. By induction, $S_i \neq \emptyset$, for $i \leq n$.

(iii) → (ii). Assume that $S_i \neq \emptyset$ for $1 \leq i \leq n$. We choose a state $s_n \in S_n$ and inductively construct a trace backward. Assume that $s_i \in S_i$. From the definition of S_i, if follows that $s_i \in Img(S_{i-1}, R) \cap h^{-1}(\widehat{s_i})$ and S_{i-1} is not empty. Select s_{i-1} from S_{i-1} such that $(s_{i-1}, s_i) \in R$. There is such and s_{i-1} since every state in s_i is a successor of some state in S_{i-1}. From the definition of S_{i-1}, $S_{i-1} \subseteq h^{-1}(\widehat{s_{i-1}})$. Hence, $s_{i-1} \in h^{-1}(\widehat{s_{i-1}})$. By induction, $s_1 \in S_1 = h^{-1}(\widehat{s_1}) \cap I$. Therefore, the trace $\langle s_1, \ldots, s_n \rangle$ that we have constructed satisfies the definition of $h^{-1}(\widehat{T})$. Thus, $h^{-1}(\widehat{T})$ is not empty. □

Suppose that condition (iii) of Lemma 4.10 is violated, and let i be the largest index such that $S_i \neq \emptyset$. Then $\widehat{s_i}$ is called the *failure state* of the spurious counterexample $\widehat{T}$.

Example 4.11. Consider a program with only one variable with domain $D = \{1, \ldots, 12\}$. 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 6; small dots denote nonreachable 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 4.10, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that $Img(S_3, R)$ and therefore S_4 are both empty. Thus, $\widehat{s_3}$ is the failure state.

It follows from Lemma 4.10 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 7 computes this number and the set of states S_{i-1}; the states in S_{i-1} are called *dead-end* states. After the detection of the dead-end states, we proceed to the refinement step (see Section 4.4). On the other hand, if the conditions stated in Lemma 4.10 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

4.3.2. *Identification of Spurious Loop Counterexamples.* Now we consider the case when the counterexample $\widehat{T}$ includes a loop, which we write as $\langle \widehat{s_1}, \ldots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \ldots, \widehat{s_n} \rangle^{\omega}$. The loop starts at the abstract state $\widehat{s_{i+1}}$ and ends at

Algorithm SplitPATH($\widehat{T}$)

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

FIG. 7. SplitPATH checks if an abstract path is spurious.

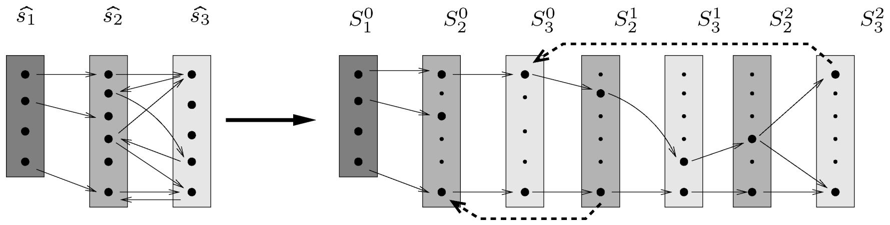


FIG. 8. A loop counterexample, and its unwinding.

$\widehat{s_n}$. Since this case is more complicated than the path counterexamples, we first present an example in which some of the typical situations occur.

Example 4.12. We consider a loop $\langle\widehat{s_1}\rangle\langle\widehat{s_2}, \widehat{s_3}\rangle^{\omega}$ as shown in Figure 8. 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, 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 theorem however shows that a polynomial number of unwindings is sufficient. (In Section 6, we indicate further practical improvements.) Let min be the minimum size of all abstract states in the loop, that is, $\min = \min_{i+1 \leq j \leq n} |h^{-1}(\widehat{s_j})|$. $\widehat{T}_{\text{unwind}}$ denotes the finite abstract path $\langle\widehat{s_1}, \ldots, \widehat{s_i}\rangle\langle\widehat{s_{i+1}}, \ldots, \widehat{s_n}\rangle^{\min+1}$, that is, the path obtained by unwinding the loop part of $\widehat{T}$ min times.

THEOREM 4.13. *The following are equivalent:*

(i) $\widehat{T}$ *corresponds to a concrete counterexample.*

(ii) $h_{\text{path}}^{-1}(\widehat{T}_{\text{unwind}})$ *is not empty.*

PROOF. We start with some easy observations. Let $\widehat{T} = \langle \widehat{s_1}, \ldots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \ldots, \widehat{s_n} \rangle^\omega$ be an abstract loop counterexample. For an index j, let j^+ denote its successor index in the counterexample, that is, $j^+ = j + 1$ for $j < n$, and $n^+ = i + 1$.

Recall that R is the transition relation of the Kripke structure. By definition, the elements of $h^{-1}(\widehat{T}_{\text{unwind}})$ are all the finite R-paths P of the form

$$\left\langle a_1, \ldots, a_i,\ b^1_{i+1}, \ldots, b^1_n, \ldots, b^{\min+1}_{i+1}, \ldots, b^{\min+1}_n \right\rangle \qquad (*)$$

for which the following two properties hold:

(i) $a_j \in h^{-1}(\widehat{s_j})$ for all $j \in [1, i]$, and
(ii) $b^k_j \in h^{-1}(\widehat{s_j})$ for all $(j, k) \in [i + 1, n] \times [1, \min + 1]$.

Each such path P has length $L := i + (\min + 1) \times (n - i)$, and we can equivalently write P in the form

$$\langle d_1, \ldots, d_L \rangle \qquad (**)$$

with the properties

(i) $d_1 \in h^{-1}(\widehat{s_1})$, and
(ii) for all $j < L$, if $d_j \in h^{-1}(\widehat{s_k})$, then $d_{j+1} \in h^{-1}(\widehat{s_{k^+}})$.

Recall that min was defined to be the size of the smallest abstract state in the loop, that is, $\min\{|h^{-1}(\widehat{s^{i+1}}|, \ldots, |h^{-1}(\widehat{s^n})|\}$, and let M be the index of an abstract state $\widehat{s_M}$ such that, $|h^{-1}(\widehat{s_M})| = \min$. (Such a state must exist, because the minimum must be obtained somewhere.)

(i) → (ii). Suppose there exists a concrete counterexample. Since the counterexample contains a loop, there exists an *infinite* $R-$path $I = \langle c_1, \ldots, \rangle$ such that $c_1 \in h^{-1}(\widehat{s_1})$, and for all j, if $c_j \in h^{-1}(\widehat{s_k})$, then $c_{j+1} \in h^{-1}(\widehat{s_{k^+}})$. In accordance with $(**)$, the finite prefix $\langle c_1, \ldots, c_L \rangle$ of I is contained in $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$, and thus, $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$ is not empty.

(ii) → (i). Suppose that $h^{-1}_{\text{path}}(\widehat{T}_{\text{unwind}})$ contains a finite R-path P.

CLAIM. *There exists a state that appears at least twice in P.*

PROOF OF CLAIM. Suppose P is in form $(*)$. Consider the states $b^1_M, b^2_M, \ldots, b^{\min+1}_M$. By $(*)$, all b^k_M are contained in $h^{-1}(\widehat{s_M})$. By definition of M, however, $h^{-1}(\widehat{s_M})$ contains only min elements, and thus there must be at least one repetition in the sequence $b^1_M, b^2_M, \ldots, b^{\min+1}_M$. Therefore, there exists a repetition in the finite R-path P, and the claim is proved. □

Let us now write P in form $(**)$, that is, $P = \langle d_1, \ldots, d_L \rangle$, and let a repetition be given by two indices $\alpha < \beta$, such that $d_\alpha = d_\beta$. Because of the repetition, there must be a transition from $d_{\beta-1}$ to d_α, and therefore, d_α is the successor state of $d_{\beta-1}$ in a cycle. We conclude that $\langle d_1, \ldots, d_{\alpha-1} \rangle \langle d_\alpha, \ldots, d_{\beta-1} \rangle^\omega$ is a concrete counterexample. □

We conclude that loop counterexamples can be reduced to path counterexamples. In Figure 9, we describe the algorithm **SplitLOOP** which is an extension of

```
Algorithm SplitLOOP($\widehat{T}$)

    $min = \min\{|h^{-1}(\widehat{s_{i+1}})|, \ldots, |h^{-1}(\widehat{s_n})|\}$
    $\widehat{T}_{\text{unwind}} = \textbf{unwind}(\widehat{T}, min + 1)$
    Compute $j$ and $S_{\text{prev}}$ as in SplitPATH($\widehat{T}_{\text{unwind}}$)
    $k := \textbf{LoopIndex}(j)$
    $p := \textbf{LoopIndex}(j + 1)$
    output $S_{\text{prev}}, k, p$
```

FIG. 9. SplitLOOP checks if an abstract loop is spurious.

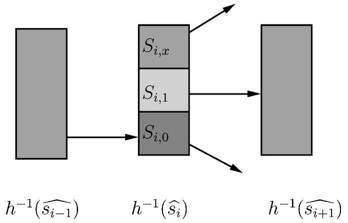


FIG. 10. Three sets $\mathcal{S}_{\text{D}}$, $\mathcal{S}_{\text{B}}$, and $\mathcal{S}_{\text{I}}$.

SplitPATH. In the algorithm, $\widehat{T}_{\text{unwind}}$ is computed by the subprogram **unwind**. The subprogram **LoopIndex**(j) computes the index of the abstract state at position j in the unwound counterexample $\widehat{T}_{\text{unwind}}$, that is, for $j \leq n$, **LoopIndex** outputs j, and for $j > n$, **LoopIndex** outputs $[(j - (i + 1) \bmod (n - i)] + (i + 1)$.

If the abstract counterexample is spurious, then the algorithm **SplitLOOP** outputs a set S_{prev} and indices k, p, such that the following conditions hold (see Figure 10):

(1) The states in S_{prev} correspond to the abstract state $\widehat{s_p}$, that is, $S_{\text{prev}} \subseteq h^{-1}(\widehat{s_p})$.
(2) All states in S_{prev} are reachable from $h^{-1}(\widehat{s_1}) \cap I$.
(3) k is the successor index of p within the loop, that is, if $p = n$ then $k = i + 1$, and otherwise $k = p + 1$.
(4) There is no transition from a state in S_{prev} to $h^{-1}(\widehat{s_k})$, that is, $Img(S_{\text{prev}}, R) \cap h^{-1}(\widehat{s_k})$ is empty.
(5) Therefore, $\widehat{s_p}$ is the failure state of the loop counterexample.

Thus, the final situation encountered is indeed very similar to the case of path counterexamples. Note that the nontrivial feature of the algorithm **SplitLOOP** is the fact that only min unwindings of the loop are necessary.

Remark 4.14. While the procedure of this section gives an exact picture of the situation to be solved, more efficient algorithms are used in our practical implementation (see Section 6).

4.4. REFINING THE ABSTRACTION. In this section, we explain how to refine an abstraction to eliminate the spurious counterexample. Recall the discussion concerning Figure 3 in Section 3 where we identified deadend states, bad states, and irrelevant states.

First, we will consider the case when the counterexample $\widehat{T} = \langle \widehat{s_1}, \ldots, \widehat{s_n} \rangle$ is a path. Let us return to a previous example for a closer investigation of failure states.

Example 4.15. Recall that in the spurious counterexample of Figure 6, the abstract state $\widehat{3}$ was the **failure state**. There are three types of concrete states in the failure state $\widehat{3}$:

(i) The **dead-end state** 9 is reachable, but there are no outgoing transitions to the next state in the counterexample.

(ii) The **bad state** 7 is not reachable but outgoing transitions cause the spurious counterexample. The spurious counterexamples is caused by the bad state.

(iii) The **irrelevant state** 8 is neither dead-end nor bad.

The goal of the refinement methodology described in this section is to refine h so that the dead-end states and bad states do not correspond to the *the same* abstract state. Then the spurious counterexample will be eliminated.

If $\widehat{T}$ does not correspond to a real counterexample, by Lemma 4.10 (iii) there always exists a set S_i of dead-end states, that is, $S_i \subseteq h^{-1}(\widehat{s_i})$ 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 $h^{-1}(\widehat{s_1}) \cap I$. Moreover, the set S_i of dead-end states can be obtained as the output S_{prev} of **SplitPATH**. 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 *bad* 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 $h^{-1}(\widehat{s_{i+1}})$, and thus the set of bad states is not empty. We partition $h^{-1}(\widehat{s_i})$ into three subsets $\mathcal{S}_{\text{D}}$, $\mathcal{S}_{\text{B}}$, and $\mathcal{S}_{\text{I}}$ as follows:

Name	*Partition*	*Definition*
dead-end states	$\mathcal{S}_{\text{D}}$	S_i
bad states	$\mathcal{S}_{\text{B}}$	$\{s \in h^{-1}(\widehat{s_i}) \mid \exists s' \in h^{-1}(\widehat{s_{i+1}}).R(s, s')\}$
irrelevant states	$\mathcal{S}_{\text{I}}$	$h^{-1}(\widehat{s_i}) \setminus (\mathcal{S}_{\text{D}} \cup \mathcal{S}_{\text{B}})$

Intuitively, $\mathcal{S}_{\text{D}}$ denotes the set of dead-end states, that is, states in $h^{-1}(\widehat{s_i})$ that are reachable from initial states. $\mathcal{S}_{\text{B}}$ denotes the set of bad states, that is, those 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 $\mathcal{S}_{\text{B}}$ cannot be empty since we know that there is a transition from $h^{-1}(\widehat{s_i})$ to $h^{-1}(\widehat{s_{i+1}})$. $\mathcal{S}_{\text{I}}$ denotes the set of irrelevant states, that is, states that are not reachable from initial states, and do not have a transition to a state in $h^{-1}(\widehat{s_{i+1}})$. Since $\mathcal{S}_{\text{B}}$ 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 $\mathcal{S}_{\text{D}}$ and $\mathcal{S}_{\text{B}}$, that is, we need an abstraction function, in which no abstract state simultaneously contains states from $\mathcal{S}_{\text{D}}$ and from $\mathcal{S}_{\text{B}}$.

It is natural to describe the needed refinement in terms of equivalence relations: Recall that $h^{-1}(\widehat{s_i})$ 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 $\equiv_j$. Formally, we say that $\equiv'$ is a refinement of $\equiv$ if for all $j \in [1, m]$ it holds that $\equiv'_j \subseteq \equiv_j$, and write $\equiv' \subseteq \equiv$ in this case. We assume that all refining relations are supersets of the equality relation $=$ given by $\{(s, s) \mid s \in S\}$.

	3	4	5
7	1	x	x
8	0	x	1
9	x	0	0

Equivalence Class

	3/4	5
7	1	x
8	0	1
9	0	0

Refinement (a)

	3	**4/5**
7/9	1	0
8	0	1

Refinement (b)

FIG. 11. Two possible refinements of an Equivalence Class.

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, that is, the separating refinement with the smallest size.

Example 4.16. Assume that we have two variables v_1, v_2. The failure state corresponds to *one equivalence class* $E_1 \times E_2$, where $E_1 = \{3, 4, 5\}$ and $E_2 = \{7, 8, 9\}$. In Figure 11, dead-end states $\mathcal{S}_{\mathrm{D}}$ are denoted by 0, bad states $\mathcal{S}_{\mathrm{B}}$ by 1, and irrelevant states by x.

Let us consider two possible partitions of $E_1 \times E_2$:

—Case (a) : $\{(3, 4), (5)\} \times \{(7), (8), (9)\}$ (6 classes)
—Case (b) : $\{(3), (4, 5)\} \times \{(7, 9), (8)\}$ (4 classes)

Clearly, case (b) generates a coarser refinement than case (a). It can be easily checked that no other refinement is coarser than (b).

In general, the problem of finding the coarsest refinement problem is computationally intractable. The proof of the following result is given in Section 5.

THEOREM 4.17. *The problem of finding the coarsest refinement is NP-hard.*

We therefore need to obtain a good heuristics for abstraction refinement. Inspecting the proof of Theorem 4.17, we see that the hardness proof heavily exploits the existence of irrelevant states. We will therefore first consider the case when $\mathcal{S}_{\mathrm{I}}$ is empty.

In this case, we show that there exists a unique coarsest refinement which can be computed in polynomial time by a symbolic algorithm. Consider the algorithm **PolyRefine** of Figure 12. The algorithm assumes a composite abstraction function with equivalence relations $\equiv_1, \ldots, \equiv_m$ over domain $S = D_1 \times \cdots \times D_m$. States are described by m-tuples $(d_1, \ldots, d_m)$. Given a set $X \subseteq S$, an index $j \in [1, m]$, and an element $a \in D_j$, the projection set $\mathrm{proj}(X, j, a)$ is given by

$$\begin{aligned}\mathrm{proj}(X, j, a) = \{&(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \mid \\ &(d_1, \ldots, d_{j-1},\ a,\ d_{j+1}, \ldots, d_m) \in X\}.\end{aligned}$$

Algorithm PolyRefine

```
for $j := 1$ to $m$
    $\equiv'_j := \equiv_j$
    for every $a, b \in E_j$
        if $\mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)$
            then $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$
```

FIG. 12. The algorithm **PolyRefine**.

The special cases of $j = 1$ and $j = m$ are defined in the obvious way. The algorithm computes new equivalence relations $\equiv'_1, \ldots, \equiv'_j$. Here, we view equivalence relations as sets of pairs of states, as subsets of $S \times S$.

The following lemma shows that the condition $\mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)$ in the algorithm is *necessary*.

LEMMA 4.18. *Suppose that $\mathcal{S}_{\mathrm{I}} = \emptyset$. If there exist $a, b \in D_j$ such that* $\mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)$*, then every refinement $\equiv' \subseteq \equiv$ must distinguish a and b, that is, $a \not\equiv'_j b$.*

PROOF. We show the following equivalent statement: If there exist $a, b \in D_j$ such that $\mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)$ and $a \equiv'_j b$, then $\equiv'$ does **not** separate $\mathcal{S}_{\mathrm{D}}$ and $\mathcal{S}_{\mathrm{B}}$. Without loss of generality, assume that there exists $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \in \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \setminus \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)$. According to the definition of $\mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a)$, $s_1 = (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in \mathcal{S}_{\mathrm{D}}$ and $s_2 = (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \notin \mathcal{S}_{\mathrm{D}}$. Since $\mathcal{S}_{\mathrm{I}} = \emptyset$, it follows that $s_2 \in \mathcal{S}_{\mathrm{B}}$.

It is easy to see that $s_1 \equiv' s_2$ because $a \equiv'_j b$ by assumption, and $d_i \equiv'_i d_i$ for all i. We conclude that $\equiv'$ does not separate $\mathcal{S}_{\mathrm{D}}$ and $\mathcal{S}_{\mathrm{B}}$. □

Next we show that indeed the necessary condition is also sufficient to obtain a refinement.

LEMMA 4.19. *When $\mathcal{S}_{\mathrm{I}} = \emptyset$, the relation $\equiv'$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv$ and separates $\mathcal{S}_{\mathrm{D}}$ and $\mathcal{S}_{\mathrm{B}}$.*

PROOF. The algorithm **PolyRefine** computes

$$\begin{aligned}
\equiv'_j &= \equiv_j - \{(a, b) : \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b) \\
&= \{(a, b) : a \equiv_j b \wedge \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) = \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)\} \\
&= \{(a, b) : a \equiv_j b \wedge \forall d_1, \ldots, d_m. \\
&\qquad\qquad (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in \mathcal{S}_{\mathrm{D}} \leftrightarrow \\
&\qquad\qquad (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \in \mathcal{S}_{\mathrm{D}}\}
\end{aligned}$$

Using the last identity of this equation, reflexivity, symmetry and transitivity of each $\equiv'_j$ can be easily observed. Hence $\equiv'$ is an equivalence relation.

Now, we show that $\equiv'$ is a correct refinement, that is, that for all dead-end states $d \in \mathcal{S}_{\mathrm{D}}$ and bad states $b \in \mathcal{S}_{\mathrm{B}}$ it holds that $d \not\equiv' b$. Let b and d be such states, and assume towards a contradiction, that $b \equiv' d$. Recall that b and d correspond to the same abstract failure state $\widehat{s_i}$, hence $h(b) = h(d) = \widehat{s_i}$, and $b \equiv d$.

By construction, b and d have the form $b = (b_1, \ldots, b_m)$, $d = (d_1, \ldots, d_m)$ where for all $i \in [1, m]$, b_i and d_i are in D_i. Consider the sequence $x_1, \ldots, x_{m+1}$

of states constructed as follows:

$$\begin{aligned} x_1 &= (b_1, b_2, \ldots, b_m) = b \\ x_2 &= (d_1, b_2, \ldots, b_m) \\ &\vdots \\ x_m &= (d_1, \ldots, d_{m-1}, b_m) \\ x_{m+1} &= (d_1, \ldots, d_m) = d. \end{aligned}$$

All x_i are concrete states corresponding to the failure state, because by assumption $h(b) = h(d)$, hence $h_i(b_i) = h_i(d_i)$ for all $1 \leq i \leq m$. Thus, $h(x_j) = (h(b_1), \ldots, h(b_m)) = \widehat{s_i}$. In particular, all x_i are equivalent with respect to $\equiv$.

Let us consider $b = x_1$ and x_2. Since $b \equiv' d$, we know that $b_1 \equiv'_1 d_1$. By definition of $\equiv'_1$, this means that $\text{proj}(\mathcal{S}_\text{D}, 1, b_1) = \text{proj}(\mathcal{S}_\text{D}, 1, d_1)$, that is, for all $d_2, \ldots, d_m$ it holds that

$$x_1 = (b_1, b_2, \ldots, b_m) \in \mathcal{S}_\text{D} \quad \text{iff} \quad x_2 = (d_1, b_2, \ldots, b_m) \in \mathcal{S}_\text{D}.$$

Analogously, let us now consider any two neighboring states x_i, x_{i+1} in this sequence. Then the states x_i and x_{i+1} only differ in their ith component. As above, we conclude that $\text{proj}(\mathcal{S}_\text{D}, i, b_i) = \text{proj}(\mathcal{S}_\text{D}, i, d_i)$, and therefore

$$x_i \in \mathcal{S}_\text{D} \quad \text{iff} \quad x_{i+1} \in \mathcal{S}_\text{D}.$$

By this chain of equivalences it follows that $b = x_1 \in \mathcal{S}_\text{D}$ iff $d = x_{m+1} \in \mathcal{S}_\text{D}$. This contradicts our assumption that $b \in \mathcal{S}_\text{B}$ and $d \in \mathcal{S}_\text{D}$. □

COROLLARY 4.20. *For $\mathcal{S}_\text{I} = \emptyset$, the equivalence relation $\equiv'$ computed by* **PolyRefine** *is the coarsest refinement of $\equiv$ which separates $\mathcal{S}_\text{D}$ and $\mathcal{S}_\text{B}$.*

PROOF. By Lemma 4.19, we know that $\equiv'$ is a correct refinement. By Lemma 4.18, all correct refinements are refinements of $\equiv'$. □

Remark 4.21. Note that in symbolic presentation, the projection operation $\text{proj}(\mathcal{S}_\text{D}, j, a)$ amounts to computing a generalized cofactor, which can be easily done by standard BDD methods. Given a function $f\colon D \to \{0, 1\}$, 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 $\mathcal{S}_\text{D}$ is represented by a function $f_{\mathcal{S}_\text{D}}\colon D \to \{0, 1\}$, and therefore, the projection $\text{proj}(\mathcal{S}_\text{D}, j, a)$ of $\mathcal{S}_\text{D}$ to value a of the jth component corresponds to a cofactor of $f_{\mathcal{S}_\text{D}}$.

Let us now return to the general case where $\mathcal{S}_\text{I}$ is not empty.

COROLLARY 4.22. *If $\mathcal{S}_\text{I}$ is not empty, the relation $\equiv'$ computed by* **PolyRefine** *is an equivalence relation refining $\equiv$ which separates $\mathcal{S}_\text{D}$ and $\mathcal{S}_\text{B} \cup \mathcal{S}_\text{I}$.*

This gives rise to the following heuristic relaxation that we use in our implementation:

REFINEMENT HEURISTICS. *We use the algorithm* **PolyRefine** *to find the coarsest refinement that separates the sets $\mathcal{S}_\text{D}$ and $\mathcal{S}_\text{B} \cup \mathcal{S}_\text{I}$. The equivalence relation computed* *by* **PolyRefine** *in this manner is in general not optimal, but it is a correct refinement which separates $\mathcal{S}_{\mathrm{D}}$ and $\mathcal{S}_{\mathrm{B}}$, and eliminates the spurious counterexample.*

Remark 4.23. While this heuristic has given good results in our practical experiments, there is some flexibility here how to use the states in $\mathcal{S}_{\mathrm{I}}$. In particular, one can also separate $\mathcal{S}_{\mathrm{D}} \cup \mathcal{S}_{\mathrm{I}}$ from $\mathcal{S}_{\mathrm{B}}$, or associate the elements of $\mathcal{S}_{\mathrm{I}}$ heuristically to either $\mathcal{S}_{\mathrm{D}}$ or $\mathcal{S}_{\mathrm{B}}$.

Since in accordance with Theorem 4.13, the algorithm **SplitLOOP** for loop counterexamples works analogously to **SplitPATH**, the refinement procedure for spurious loop counterexamples works analogously, that is, it uses **SplitLOOP** to identify the failure state, and **PolyRefine** to obtain a heuristic refinement.

Our refinement procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the ACTL* 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 4.24. *Given a model M and an* ACTL* *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. *Complexity of Optimal Abstraction Refinement*

Recall that, in Figure 11, we have visualized the special case of two variables and two equivalence relations in terms of matrices. In order to formally capture this visualization, let us define the **Matrix Squeezing** problem.

Definition 5.1. **Matrix Squeezing.** Given a finite matrix with entries $0, 1, x$ and a constant Γ, is it possible to construct a matrix with $\leq \Gamma$ entries by iterating the following operations:

(1) Merging two compatible rows.

(2) Merging two compatible columns.

Two columns are *compatible*, if there is no position, where one column contains 1 and the other column contains 0. All other combinations are allowed, that is, x does not affect compatibility. *Merging* two compatible columns means replacing the columns by a new one which contains 1 at those positions where at least one of the two columns contained 1, and 0 at those positions, where at least one of the two columns contained 0. For rows, the definitions are analogous.

Since this is a special case of the refinement problem, it is sufficient to show NP-hardness for **Matrix Squeezing**. Then it follows that the refinement problem is NP-hard, too, and thus Theorem 4.17 is proved.

As mentioned **Matrix Squeezing** is easy to visualize: If we imagine the symbol x to be transparent, then merging two columns can be thought of as putting the two (transparent) columns on top of each other. **Column Squeezing** is a variant of **Matrix Squeezing**, where only columns can be merged:

Definition 5.2. **Column Squeezing.** Given a finite matrix with entries $0, 1, x$ and a constant Δ, is it possible to construct a matrix with $\leq \Delta$ *columns* by iterated merging of *columns*?

The proof will be by reduction from problem GT15 in Garey and Johnson [1979]:

Definition 5.3. **Partition Into Cliques.** Given an undirected graph (V, E) and and a number $K \geq 3$, is there a partition of V into $k \leq K$ classes, such that each class induces a clique on (V, E)?

THEOREM 5.4 [KARP 1972]. **Partition Into Cliques** *is NP-complete.*

THEOREM 5.5. **Column Squeezing** *is NP-complete.*

PROOF. Membership is trivial. Let us consider hardness. We reduce **Partition Into Cliques** to **Column Squeezing**. Given a graph (V, E) and a number K, we have to construct a matrix M and a number Δ such that M can be squeezed to size $\leq \Delta$ iff (V, E) can be partitioned in $\leq K$ cliques.

We construct a $(|V|, |V|)$ matrix $(a_{i,j})$ which is very similar to the adjacency matrix of (V, E):

$$a_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } (i, j) \notin E, i \neq j \\ x & \text{if } (i, j) \in E, i \neq j \end{cases}$$

Assume without loss of generality that $V = \{1, \ldots, n\}$. Then it is not hard to see that for all $i, j \in V$, columns i and j are compatible iff $(i, j) \in E$, since by construction we included 0 in the matrix only to forbid merging of columns.

By construction, (V, E) contains a clique C with vertices $c_1, \ldots, c_l$, if the columns $c_1, \ldots, c_l$ can all be merged into one. (Note however that compatibility is not a transitive relation. The existence of a clique only guarantees that all positions in all columns in the clique are compatible with the corresponding positions in the other columns in the clique.)

Thus, (V, E) can be partitioned into $\leq K$ cliques, iff the columns of $(a_{i,j})$ can be merged into $\leq K$ columns. Setting $\Delta = K$ concludes the proof. □

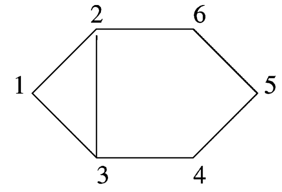
THEOREM 5.6. **Matrix Squeezing** *is NP-complete.*

PROOF. Membership is trivial. We show hardness by reducing **Column Squeezing** to **Matrix Squeezing**. For an integer n, let $|bin(n)|$ denote the size of the binary representation of n. Given an (n, m) matrix M and a number Δ, it is easy to construct an $(n + 1, m + |bin(m - 1)|)$ matrix $B(M)$ by adding additional columns to A in such a way that

(i) all rows of $B(M)$ become incompatible, and
(ii) no new column is compatible with any other (new or old) column.

An easy construction to obtain this is to concatenate the rows of M with the binary encodings of the numbers $0, \ldots, m - 1$ over alphabet $\{0, 1\}$, such that the ith row is concatenated with the binary encoding of the number $i - 1$. Since any two different binary encodings are distinguished by at least one position, no two rows are compatible. In addition, we add an $n + 1$st row that contains 1 on positions in the original columns, and 0 on positions in the new columns. Thus, in matrices of the form $B(M)$, only columns that already appeared in M (with an additional 0 symbol below) can be compatible.

It remains to determine Γ. We set $\Gamma := (\Delta + |bin(m - 1)|) \times (n + 1)$. The term $|bin(m - 1)|$ takes into account that we have added $|bin(m - 1)|$ columns, and the



	1	2	3	4	5	6
1	1	x	x	0	0	0
2	x	1	x	0	0	x
3	x	x	1	x	0	0
4	0	0	x	1	x	0
5	0	0	0	x	1	x
6	0	x	0	0	x	1

Column Squeezing

	1	2	3	4	5	6	7	8	9
1	1	x	x	0	0	0	0	0	0
2	x	1	x	0	0	x	0	0	1
3	x	x	1	x	0	0	0	1	0
4	0	0	x	1	x	0	0	1	1
5	0	0	0	x	1	x	1	0	0
6	0	x	0	0	x	1	1	0	1
7	0	0	0	0	0	0	1	1	1

Matrix Squeezing

FIG. 13. An instance of **Partition into Cliques**, and its reduction images.

factor $(n + 1)$ takes into account that Δ is counting columns, while Γ is counting matrix entries. □

Example 5.7. Figure 13 demonstrates how a graph instance is reduced to a matrix instance. Note for example that $\{1, 2, 3\}$ is a clique in the graph, and therefore, the columns $1, 2, 3$ of the **Column Squeezing** problem are compatible. In the **Matrix Squeezing Instance**, Columns 7, 8, 9 enforce that no rows can be merged. Row 7 guarantees that columns 7, 8, 9 can not be merged with columns $1, \ldots, 6$ or with themselves.

Remark 5.8. It is highly unlikely that there are PTIME *approximation* algorithms for computing the size of an optimal abstraction refinement. [Bellare et al. 2003] have shown that no polynomial time algorithm can approximate the chromatic number of a graph (V, E) within $|V|^{1/7-\epsilon}$ for any $\epsilon > 0$ unless $\mathrm{P} = \mathrm{NP}$. It is easy to see that there is a metric reduction between the chromatic number problem and the problem of computing the size of an optimal abstraction refinement for two variable clusters. (Note that **Partition Into Cliques** is reducible to **Graph Coloring** by a trivial reduction which maps a graph to its complement.) Thus, nonapproximability of optimal abstraction refinement follows. Under the assumption that coRP is different from RP, an even stronger nonapproximability result for computing chromatic number has been shown by Feige and Kilian [1996]. A rigorous proof of these observations exceeds the scope of this article and is omitted. Since there are no approximation algorithms, we have to rely on heuristics to solve the abstraction refinement problem.

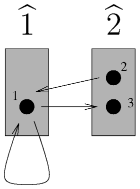


FIG. 14. A spurious loop counterexample $\langle\widehat{1},\widehat{2}\rangle^{\omega}$.

6. *Performance Improvements*

The symbolic methods described in Section 4.2.3 can be directly implemented using BDDs. Our implementation uses additional heuristics that are outlined in this section.

Two-phase Refinement Algorithms. Consider the spurious loop counterexample $\widehat{T} = \langle\widehat{1},\widehat{2}\rangle^{\omega}$ of Figure 14. 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 $S_{\text{local}} := (\bigcup_{1\le i\le n} h^{-1}(\widehat{s}_i))$ of the counterexample and use the standard fixpoint computation for temporal formulas (see, e.g., Clarke et al. [1999a]) 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. Recall from the discussion on early approximation in Section 3.1 that the transition relation R of a large model is often partitioned [Clarke et al. 1999a] into transition relations $R_1, \ldots, R_n$, where each R_i is the transition relation for a single variable in the system. The abstract transition relation is over-approximated by the relation

$$\mathcal{T}(R) = (R_1)_h \wedge \cdots \wedge (R_n)_h \supseteq R_h,$$

yielding a Kripke structure $\tilde{M}_{\mathcal{T}}$. The transitions in the set difference $\mathcal{T} - R_h$ are called *virtual transitions.*

As illustrated by Example 3.7, the disadvantage of early approximation is that the virtual transitions in $\tilde{M}_{\mathcal{T}}$ may obscure important behavior, cf. also Clarke et al. [1999].

Recall from Section 4.2.3 that in our approach the composite abstraction function is defined for each variable cluster independently. Therefore, we can use a heuristic to determine 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 over-approximation and memory usage. Moreover, the overall method presented in our paper remains complete with this approximation.

Assume that we have k variable clusters $VC_1, \ldots, VC_k$, and that $T_1, \ldots, T_k$ are the transition relations corresponding to the variable clusters.

Let T_i^{combined} be the abstraction of T_i according to the heuristics, that is, if we do not abstract VC_i, then $T_i^{\text{combined}} = T_i$, otherwise, we set $T_i^{\text{combined}} = (T_i)_h$, and define $\mathcal{T}^{\text{combined}}(R) = (T_1)^{\text{combined}} \wedge \cdots \wedge (T_k)^{\text{combined}}$. We call this abstraction *combined approximation*. The proof of the following lemma is transparent.

LEMMA 6.1. *Combined approximation is intermediate between existential abstraction and early approximation, that is,*

$$R \subseteq \mathcal{T}^{\text{combined}}(R) \subseteq \mathcal{T}(R).$$

Our method remains complete, because during the symbolic simulation of the counterexample the algorithms **SplitPATH** and **SplitLOOP** treat both forms of over-approximations, that is, virtual transitions and spurious transitions, in the same way.

Abstractions for Distant Variables. In addition to the methods of Section 4.2.2, we completely abstract away 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 [Balarin and Sangiovanni-Vincentelli 1993; Kurshan 1994; Lind-Nielsen and Andersen 1999] in a similar way. However, the refinement process of the localization reduction [Kurshan 1994] can only turn a completely abstracted variable into a completely unabstracted variable, while our method uses intermediate abstraction functions.

A user-defined integer constant `far` determines which variables are close to the specification φ. The set `NEAR` of near variables contains those variables whose distance from the specification in the dependency graph is at most `far`, and `FAR` $=$ $var(P) -$ `NEAR` is the set of far variables. For variable clusters without far variables, the abstraction function remains unchanged. For variable clusters with far variables, their far variables are completely abstracted away, and their near variables remain unabstracted. Note that the initial abstraction for variable clusters with far variables looks similar as in the localization reduction.

7. *Experimental Results*

We have implemented our methodology in NuSMV [Cimatti et al. 1998] which uses the CUDD package [Somenzi 2003] for symbolic representation. We performed two sets of experiments. One set includes four benchmark designs and three industrial designs from Synopsys. The other was performed on an industrial design of a multimedia processor from Fujitsu [1996]. All the experiments were carried out on a 200 MHz PentiumPro PC with 1 GB RAM memory using Linux.

7.1. EXPERIMENTS ON BENCHMARK CIRCUITS.

The first benchmark set includes four publicly available designs and three industrial designs. Properties of these designs are described in Table I. In the table, the second column (#Var) shows the number of symbolic variables in the designs, while the third column (#Reg)

TABLE I. PROPERTIES OF THE GIVEN BENCHMARK DESIGNS

Design	#Var	#Reg	#Spec
gigamax	10	16	1
guidance	40	55	8
waterpress	6	21	8
PCI bus	50	89	15
ind1	72	72	1
ind2	101	101	1
ind3	190	190	1

TABLE II. RUNNING RESULTS FOR THE BENCHMARK DESIGNS

Design	NuSMV+COI				NuSMV+ABS			
	#COI	Time	$\|TR\|$	$\|MC\|$	#ABS	Time	$\|TR\|$	$\|MC\|$
gigamax	0	0.3	8346	1822	9	0.2	13151	816
guidance	30	35	140409	30467	34–39	30	147823	10670
waterpress	0–1	273	34838	129595	4	170	38715	3335
PCI bus	4	2343	121803	926443	12–13	546	160129	350226
ind1	0	99	241723	860399	50	9	302442	212922
ind2	0	486	416597	2164025	84	33	362738	624600
ind3	0	617	584815	2386682	173	15	426162	364802

shows the corresponding number of the Boolean variables. For example, a symbolic variable whose domains has size eight corresponds to three Boolean variables. Therefore, the number of Boolean variables is always larger or equal to the number of symbolic variables. Overall, 37 specifications are considered in this benchmark.

The results for these designs are listed in Table II. Note that average time and space usage per design are reported in this table. 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. 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.

We also report the relative time and space difference between our approach and traditional cone of influence reduction in Figure 15 and Figure 16. In the figures, the x axis corresponds to the number of properties and y axis corresponds to the relative time and space difference respectively (Time(COI)/Time(Abs) and Space(COI)/Space(Abs)). Although our approach uses less than 50% more memory than the traditional cone of influence reduction to *build* the abstract transition relation, it requires one order of magnitude of memory less 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.

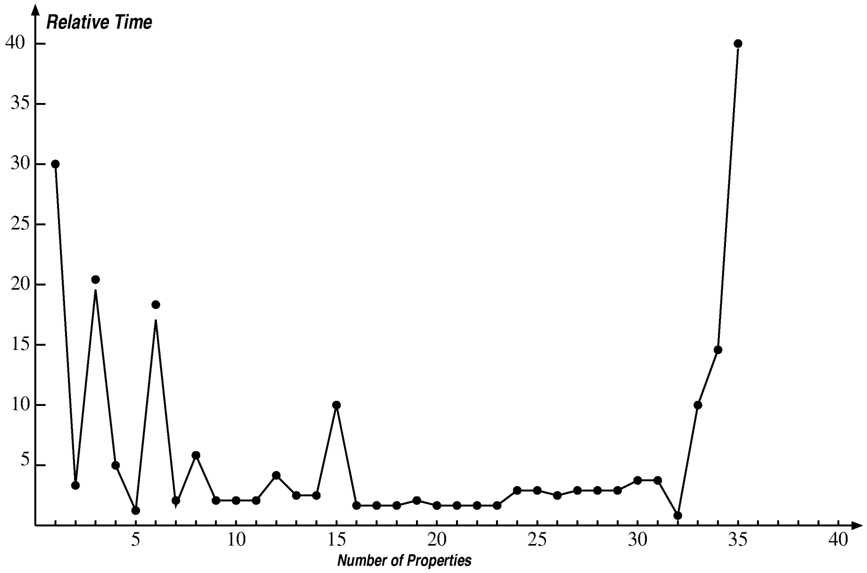


FIG. 15. The Relative Time (Time(COI)/Time(ABS)) Improvement.

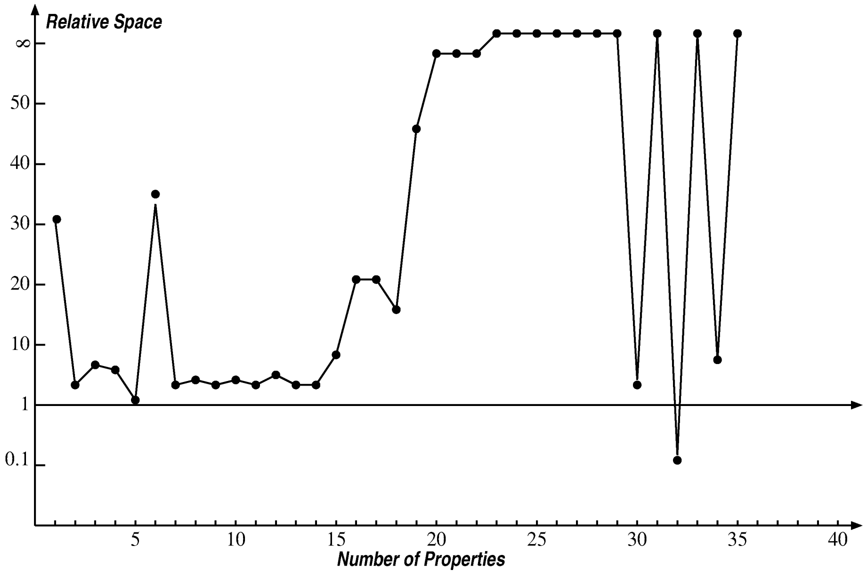


FIG. 16. The Relative Space (Space(COI)/Space(ABS)) Improvement.

7.2. DEBUGGING A MULTIMEDIA PROCESSOR. As another example, we verified a multimedia assist (MMA-ASIC) processor developed by Fujitsu [1996]. The system configuration of this processor is shown in Figure 17 [Takayama et al. 1998]. A dashed line represents a chip boundary. MM-ASIC is connected to a host CPU and external I/O units via “Bus-H”, and to SDRAMs via “Bus-R”. MM-ASIC consists of a co-processor for multimedia instructions (MMA), a graphic display controller (GDC), peripheral I/O units, and five bus bridges (BBs).

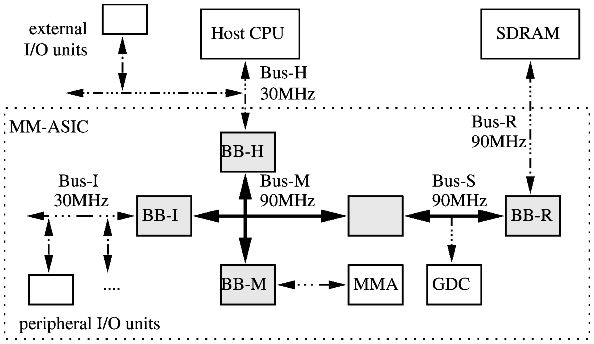


FIG. 17. Configuration of MMA-ASIC.

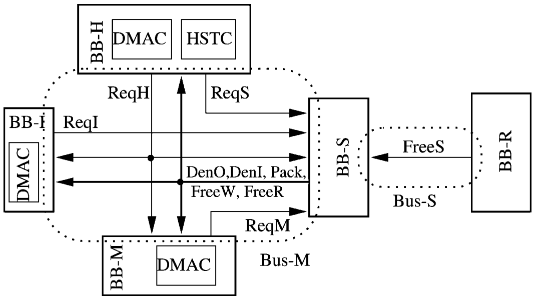


FIG. 18. Control signals on "Bus-M".

It is one of the characteristics of a system-on-chip that the design contains bus bridges, because the components of the system may have different interface protocols or different frequencies of operation. Bus bridges are used to construct a bus hierarchy according to the locality of transactions. MM-ASIC consists of the following five bus bridges.

—"BB-I" and "BB-H". These separate Bus-M from Bus-H and Bus-I, since the bus frequency of Bus-M is different from that of Bus-H and Bus-I.

—"BB-S". This separates the transactions between GDC and SDRAM from those between MMA and host CPU, since they are major transactions in MM-ASIC.

—"BB-R". This resolves the difference between the protocols of Bus-R and Bus-S.

—"Bus-M". This separates Bus-M from the local bus of MMA.

The RTL implementation of MM-ASIC is described in Verilog-HDL. The total number of lines of code is about 61,500. The verification is targeted to verify the correctness of the bus transactions. Therefore, three operational units, peripheral I/Os, MMA, and GDC are omitted. After this elimination of the units, the number of registers is reduced to about 4000. Fujitsu engineers then abstracted away the data path which is not useful for our verification task. The final description contains about 500 latches.

Figure 18 shows some control signals and controllers within bus bridges. BB-H, BB-I and BB-M contains a DMA controller "DMAC" which controls a DMA transfer between SDRAM and a component, such as an external/internal IO and MMA. BB-H contains another controller "HSTC" which controls a data transfer between a host and all components except the external IOs. BB-R asserts “FreeS” when it can accept a request from Bus-S. BB-S asserts “FreeW” (“FreeR”) when it can accept a write (read) request from Bus-M. A bus transaction on Bus-M consists of the following four phases.

—*Arbitration phase* is the first cycle when a bus master asserts a request signal. When more than one master requests, only the master with the highest priority goes into request phase in the next cycle. “ReqS”, “ReqM”, and “ReqI” are request signals on Bus-M for DMA transfer from/to SDRAM. The signals are asserted by BB-H, BB-M, and BB-I respectively. “ReqH” is a request asserted by BB-H for normal (non-DMA) data transfer. The requests are ordered by priority as follows: $ReqM \prec ReqI \prec ReqS = ReqH$.

—*Request phase* is the next cycle after the arbitration phase. A bus master passes the address and other control signals to a bus slave.

—*Ready phase* is the cycle were the data is ready to be transferred. “DenO” (“DenI”) is asserted when the write (read) data is ready to transfer in the next cycle.

—*Transfer phase* is the next cycle after the ready phase. “Pack” is asserted when the data is transferred between BB-H and a bus bridge, such as BB-M and BB-I, in the next cycle. Data is transferred between a master and a slave.

In Takayama et al. [1998], 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, that is, it may fail to prove the correctness even if the property is true. Moreover, the navigation conditions are usually not automatically generated. Since our model checker can only accept the SMV language, we translated this abstracted Verilog code into 9,500 lines of SMV code.

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 [Yang et al. 1998] and NuSMV [Cimatti et al. 1998]. 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 using three refinement steps was successfully able to verify the design. During the verification, we discovered a bug that had not been discovered before.*

8. *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 are currently applying our technique to verify other large examples. We believe that our technique is general enough to be adapted to other forms of abstraction.

There are many interesting avenues for future research: Our current work concentrates on two directions: First, we are extending the refinement methodology to full ACTL. This requires to extend the notion of counterexamples beyond paths and loops. Results in this direction are reported in Clarke et al. [2001]. Since the symbolic methods described in this paper are not tied to representation by BDDs, we are also investigating how they can be applied to recent work on symbolic model checking without BDDs [Biere et al. 1999].

Appendix

A. *Background Material*

If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \pi \models f$ means that f holds along path π in M. The relation $\models$ is defined recursively as follows (assuming that f_1 and f_2 denote state formulas and g_1 and g_2 denote path formulas):

$$\begin{array}{lcl}
M, s \models p & \Leftrightarrow & p \in L(s); \\
M, s \models \neg f_1 & \Leftrightarrow & M, s \not\models f_1; \\
M, s \models f_1 \wedge f_2 & \Leftrightarrow & M, s \models f_1 \text{ and } M, s \models f_2; \\
M, s \models f_1 \vee f_2 & \Leftrightarrow & M, s \models f_1 \text{ or } M, s \models f_2; \\
M, s \models \mathbf{E}\, g_1 & \Leftrightarrow & \text{there exists } \pi = \langle s, \ldots \rangle \text{ such that } M, \pi \models g_1; \\
M, s \models \mathbf{A}\, g_1 & \Leftrightarrow & \text{for every } \pi = \langle s, \ldots \rangle \text{ it holds that } M, \pi \models g_1; \\
M, \pi \models f_1 & \Leftrightarrow & \pi = \langle s, \ldots \rangle \text{ and } M, s \models f_1; \\
M, \pi \models \neg g_1 & \Leftrightarrow & M, \pi \not\models g_1; \\
M, \pi \models g_1 \wedge g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ and } M, \pi \models g_2; \\
M, \pi \models g_1 \vee g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ or } M, \pi \models g_2; \\
M, \pi \models \mathbf{X}\, g_1 & \Leftrightarrow & M, \pi^1 \models g_1; \\
M, \pi \models \mathbf{F}\ g_1 & \Leftrightarrow & \exists k \geq 0, \text{ such that } M, \pi^k \models g_1; \\
M, \pi \models \mathbf{G}\ g_1 & \Leftrightarrow & \forall k \geq 0,\ M, \pi^k \models g_1 \text{ holds} \\
M, \pi \models g_1\, \mathbf{U}\ g_2 & \Leftrightarrow & \exists k \geq 0, \text{ such that } M, \pi^k \models g_2 \text{ and } \forall 0 \leq j < k, \text{ it holds} \\
 & & \text{that } M, \pi^j \models g_1. \\
M, \pi \models g_1\, \mathbf{R}\ g_2 & \Leftrightarrow & \forall j \geq 0, \text{ if for every } i < j,\ M, \pi^i \not\models g_1 \text{ then} \\
 & & \text{that } M, \pi^j \models g_1.
\end{array}$$

An *ordered binary decision diagram* (BDD) is an efficient data-structure for representing boolean functions, and moreover, BDDs support all the operations that can be performed on boolean functions. Let A be a set of propositional variables, and $\prec$ a linear order on A. Formally, an ordered binary decision diagram $\mathcal{O}$ over A is an acyclic graph (V, E) whose nonterminal vertices (*nodes*) are labeled by variables from A, and whose edges and terminal nodes are labeled by 0, 1. Each nonterminal node v has out-degree 2, such that one of its outgoing edges is labeled 0 (the *low edge* or *else-edge*), and the other is labeled 1 (the *high edge* or *then-edge*). If v has label a_i and the successors of v are labeled a_j, a_k, then $a_i \prec a_j$ and $a_i \prec a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $\prec$. Each BDD node v represents a Boolean function $\mathcal{O}_v$. The terminal nodes of $\mathcal{O}$ represent the constant functions given by their labels. A non-terminal node v with label a_i whose successors at the high and low edges are u and w respectively, defines the function $\mathcal{O}_v := (a_i \wedge \mathcal{O}_u) \vee (\neg a_i \wedge \mathcal{O}_w)$. The size of a BDD is the number of nodes of the BDD. The size of a BDD in general depends on the variable order $\prec$, and may be exponential in $|A|$. However, it is well-known [Bryant 1986; Bryant 1991] that for every variable order $\prec$ and Boolean function f there exists a *unique minimal* BDD $\mathcal{O}$ over A which represents the Boolean function f. Given any BDD for f which respects $\prec$, $\mathcal{O}$ can be computed in polynomial time. Note that $\mathcal{O}$ contains at most two terminal nodes, and no two nodes of $\mathcal{O}$ describe the same Boolean function. In practice, *shared BDDs* are used to represent several Boolean functions by different nodes in the graph. Effective algorithms for handling BDDs have been described in the literature [Bryant 1986] and highly effective BDD libraries such as CUDD [Somenzi 2001] have been developed.

ACKNOWLEDGMENTS. The authors are grateful to Dong Wang for helpful discussions about predicate abstraction, and to Christian Schallhart for advice concerning nonapproximability results.

REFERENCES

ABDULLA, P. A., ANNICHINI, A., BENSALEM, S., BOUAJJANI, A., HABERMEHL, P., AND LAKHNECH, Y. 1999. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Computer-Aided Verification (CAV)*.

BALARIN, F., AND SANGIOVANNI-VINCENTELLI, A. L. 1993. An iterative approach to language containment. In *Computer-Aided Verification (CAV)*.

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York.

BELLARE, M., GOLDREICH, O., AND SUDAN, M. 2003. Free bits, PCPs and non-approximability—towards tight results. *SIAM J. Comput. 27*, 804–915.

BENSALEM, S., BOUAJJANI, A., LOISEAUX, C., AND SIFAKIS, J. 1992. Property preserving simulations. In *Computer-Aided Verification (CAV)*.

BENSALEM, S., LAKHNECH, Y., AND OWRE, S. 1998. Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification (CAV)*.

BEREZIN, S., BIERE, A., CLARKE, E., AND ZHU, Y. 1998. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Formal Methods in Computer-Aided Design*.

BIERE, A., CIMATTI, A., CLARKE, E., FUJITA, M., AND ZHU, Y. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference*.

BJORNER, N. S., BROWNE, A., AND MANNA, Z. 1997. Automatic generation of invariants and intermediate assertions. *Theoret. Comput. Sci. 173*, 1, 49–87.

BRUNS, G., AND GODEFROID, P. 1999. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification (CAV)*.

BRYANT, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput. 35*, 8, 677–691.

BRYANT, R. E. 1991. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput. 40*, 205–213.

BURCH, J., AND DILL, D. 1994. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV)*.

BURCH, J. R., CLARKE, E. M., AND LONG, D. E. 1991. Symbolic model checking with partitioned transition relations. In *Proceedings of the 1991 International Conference on Very Large Scale Integration*, A. Halaas and P. B. Denyer, Eds. Winner of the Sidney Michaelson Best Paper Award.

BURCH, J. R., CLARKE, E. M., AND MCMILLAN, K. L. 1992. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput. 98*, 142–170.

CIMATTI, A., CLARKE, E., GIUNCHIGLIA, F., AND ROVERI, M. 1998. NuSMV: A new symbolic model checker. In *Software Tools for Technology Transfer*.

CLARKE, E., ENDERS, R., FILKORN, T., AND JHA, S. 1996. Exploiting symmetry in temporal logic model checking. *Form. Meth. Syst. Des. 9*, 1/2, 41–76.

CLARKE, E., JHA, S., LU, Y., AND WANG, D. 1999. Abstract BDDs: A technique for using abstraction in model checking. In *Correct Hardware Design and Verification Methods (CHARME)*.

CLARKE, E., LU, Y., JHA, S., AND VEITH, H. 2001. Counterexamples in model checking. Tech. Rep., Carngie Mellon University. CMU-CS-01-106.

CLARKE, E. M., AND EMERSON, E. A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1983. Automatic verification of finite-state concurrent system using temporal logic. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York.

CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. 1994. Model checking and abstraction. *ACM Trans. Prog. Lang. Syst. (TOPLAS) 16*, 5 (Sept.), 1512–1542.

CLARKE, JR., E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst. (TOPLAS) 8*, 2 (Apr.), 244–263.

COLÓN, M. A., AND URIBE, T. E. 1998. Generating finite-state abstraction of reactive systems using decision procedures. In *Computer-Aided Verification (CAV)*.

COUDERT, O., BERTHET, C., AND MADRE, J. C. 1989. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, J. Sifakis, Ed. Lecture Notes in Computer Science, vol. 407. Springer-Verlag, New York.

COUSOT, P., AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium of Programming Language*, 238–252.

COUSOT, P., AND COUSOT, R. 1999. Refining model checking by abstract interpretation. *Automat. Softw. Eng. 6*, 69–95.

DAMS, D., GERTH, R., AND GRUMBERG, O. 1997a. Abstract interpretation of reactive systems. *ACM Trans. Prog. Lang. Syst. (TOPLAS) 19*, 2.

DAMS, D. R., GRUMBERG, O., AND GERTH, R. 1993. Generation of reduced models for checking fragments of CTL. In *Computer-Aided Verification (CAV)*.

DAMS, D. R., GRUMBERG, O., AND GERTH, R. 1997b. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL*, CTL*. In *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET 94)*.

DAS, S., AND DILL, D. L. 2001. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, Los Alamitos, Calif.

DAS, S., DILL, D. L., AND PARK, S. 1999. Experience with predicate abstraction. In *Computer-Aided Verification (CAV)*.

DINGEL, J., AND FILKORN, T. 1995. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Computer-Aided Verification (CAV)*.

DWYER, M. B., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PASAREANU, C. S., ROBBY, VISSER, W., AND ZHENG, H. 2001. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*.

EMERSON, E., AND SISTLA, A. 1996. Symmetry and model checking. *Formal Methods in System Design 9(1/2)*, 105–130.

EMERSON, E., AND TREFLER, R. 1999. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Correct Hardware Design and Verification Methods (CHARME)*. Lecture Notes in Computer Science, vol. 1703. Springer-Verlag, New York, 142–156.

FEIGE, U., AND KILIAN, J. 1996. Zero knowledge and the chromatic number. In *Proceedings of the IEEE Conference on Computational Complexity (CCC)*. IEEE Computer Society Press, Los Alamitos, Calif., 278–287.

FEIGENBAUM, J., KANNAN, S., VARDI, M. Y., AND VISWANATHAN, M. 1999. Complexity of problems on graphs represented as OBDDs. *Chic. J. Theoret. Comput. Sci.*

FUJITSU. 1996. Fujitsu aims media processor at DVD. *MicroProcessor Rep.* 11–13.

FURA, D., WINDLEY, P., AND SOMANI, A. 1993. Abstraction techniques for modeling real-world interface chips. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, J.J. Joyce and C.-J.H. Seger, Eds. Lecture Notes in Computer Science, vol. 780. University of British Columbia, Springer Verlag, published 1994, Vancouver, Canada, 267–281.

GAREY, M. R., AND JOHNSON, D. S. 1979. *Computers and interactability: A guide to the theory of NP-Completeness*. W. H. Freeman and Company.

GODEFROID, P., PELED, D., AND STASKAUSKAS, M. 1996. Using partial order methods in the formal verification of industrial concurrent programs. In *Proceedings of the ISSTA'96 International Symposium on Software Testing and Analysis*. 261–269.

GOTTLOB, G., LEONE, N., AND VEITH, H. 1999. Succinctness as a source of complexity in logical formalisms. *Ann. Pure Appl. Logic 97*, 1–3, 231–260.

GOVINDARAJU, S. G., AND DILL, D. L. 1998. Verification by approximate forward and backward reachability. In *Proceedings of the International Conference of Computer-Aided Design (ICCAD)*.

GOVINDARAJU, S. G., AND DILL, D. L. 2000. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 115–119.

GRAF, S. 1994. Verification of distributed cache memory by using abstractions. In *Proceedings of Computer-Aided Verification (CAV)*.

GRAF, S., AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of Computer-Aided Verification (CAV)*.

HO, P.-H., ISLES, A. J., AND KAM, T. 1998. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of the International Conference of Computer-Aided Design (ICCAD)*.

HOJATI, R., AND BRAYTON, R. K. 1995. Automatic datapath abstraction in hardware systems. In *Proceedings of Computer-Aided Verification (CAV)*.

IP, C., AND DILL, D. 1996. Better verification through symmetry. *Form. Meth. Syst. Des. 9*, 1/2, 41–76.

JENSEN, K. 1996. Condensed state spaces for symmetrical colored petri nets. *Form. Meth. Syst. Des. 9*, 1/2, 7–40.

JONES, R. B., SKAKKEBAK, J. U., AND DILL, D. L. 1998. Reducing manual abstraction in formal verification of out-of-order execution. In *Form. Meth. Comput.-Aided Des.* 2–17.

KARP, R. 1972. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. 85–103.

KURSHAN, R. P. 1994. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ.

LEE, W., PARDO, A., JANG, J., HACHTEL, G., AND SOMENZI, F. 1996. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference of Computer-Aided Design (ICCAD)*. 76–81.

LESENS, D., AND SAÏDI, H. 1997. Automatic verification of parameterized networks of processes by abstraction. In *Proceedings of the International Workshop on Verification of Infinite State Systems (INFINITY)*. Bologna.

LIND-NIELSEN, J., AND ANDERSEN, H. R. 1999. Stepwise CTL model checking of state/event systems. In *Proceedings of Computer-Aided Verification (CAV)*.

LOISEAUX, C., GRAF, S., SIFAKIS, J., BOUAJJANI, A., AND BENSALEM, S. 1995. Property preserving abstractions for the verification of concurrent systems. *Form. Meth. Syst. Des.*, 1–36.

LONG, D. E. 1993. Model checking, abstraction and compositional verification. Ph.D. dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pa. CMU-CS-93-178.

MANNA, Z., COLN, M. C., FINKBEINER, B., SIPMA, H., AND URIBE, T. E. 1998. Abstraction and modular verification of infinite-state reactive systems. In *Proceedings of the Requirements Targeting Software and Systems Engineering (RTSE)*.

MCMILLAN, K. 1996. A conjunctively decomposed boolean representation for symbolic model checking. In *Proceedings of Computer-Aided Verification (CAV)*. 13–25.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

MCMILLAN, K. L. 1999a. Verification of infinite state systems by compositional model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME)*. 219–234.

MCMILLAN, K. L. 1999b. Verification of infinite state systems by compositional model checking. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME)*.

PARDO, A. 1997. Automatic abstraction techniques for formal verification of digital systems. Ph.D. dissertation, Dept. of Computer Science, University of Colorado at Boulder, Boulder Colo.

PARDO, A., AND HACHTEL, G. 1998. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference (DAC)*.

PELED, D. 1993. All from one, one from all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*. Lecture Notes in Computer Science, vol. 697. Springer-Verlag, New York (Elounda Crete, Greece). 409–423.

PIXLEY, C. 1990. A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the CAV Workshop (also DIMACS Tech. Report 90-31),* R. Kurshan and E. Clarke, Eds. Rutgers University, NJ.

PIXLEY, C., BEIHL, G., AND PACAS-SKEWES, E. 1991. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of the International Conference on Computer Design* (Cambridge, Mass.). 245–249.

PIXLEY, C., JEONG, S.-W., AND HACHTEL, G. D. 1992. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*. 620–623.

RUSHBY, J. 1999. Integrated formal verification: using model checking with automated abstraction, invariant generation, and theorem proving. In *Theoretical and practical aspects of SPIN model checking: 5th and 6th international SPIN workshops*.

RUSU, V., AND SINGERMAN, E. 1999. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

SAGIV, S., REPS, T. W., AND WILHELM, R. 1999. Parametric shape analysis via 3-valued logic. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

SAÏDI, H., AND SHANKAR, N. 1999. Abstract and model checking while you prove. In *Proceedings of Computer-Aided Verification (CAV)*.

SIFAKIS, J. 1983. Property preserving homomorphisms of transition systems. In *Proceedings of the 4th Workshop on Logics of Programs*.

SOMENZI, F. 2001. CUDD: CU decision diagram package. *http://vlsi.colorado.edu/ fabio/*.

TAKAYAMA, K., SATOH, T., NAKATA, T., AND HIROSE, F. 1998. An approach to verify a large scale system-on-chip using symbolic model checking. In *Proceedings of the International Conference of Computer Design*.

VAN AELTEN, F., LIAO, S., ALLEN, J., AND DEVADAS, S. 1992. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *International Conference of Computer-Aided Design (ICCAD)*.

VEITH, H. 1997. Languages represented by Boolean formulas. *Inf. Proc. Lett. 63*, 251–256.

VEITH, H. 1998a. How to encode a logical structure as an OBDD. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity (CCC)*. IEEE Computer Society, Press, Los Alamitos, Calif., 122–131.

VEITH, H. 1998b. Succinct representation, leaf languages and projection reductions. *Inf. Comput. 142*, 2, 207–236.

WOLPER, P., AND LOVINFOSSE, V. 1989. Verifying properties of large sets of processes with network invariants. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science, vol. 407. Springer-Verlag, New York.

YANG, B., BRYANT, R. E., O'HALLARON, D. R., BIERE, A., COUDERT, O., JANSSEN, G., AND R. K. RANJAN, F. S. 1998. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*. Lecture Notes in Computer Science, vol. 1522. Springer-Verlag, New York.

RECEIVED DECEMBER 2001; REVISED MAY 2003; ACCEPTED MAY 2003