Abstraction and Refinement in Model Checking

Orna Grumberg
Computer Science Department
Technion
Haifa 32000, Israel

1 Introduction

In this paper we survey abstraction and refinement in model checking. We restrict the discussion to existential abstraction which over-approximates the behaviors of the concrete model. The logics preserved under this abstraction are the universal fragments of branching-time temporal logics as well as linear-time temporal logics. For simplicity of presentation, we also restrict the discussion to abstraction functions, rather then abstraction relations. Thus, every concrete state is represented by exactly one abstract state. An abstract state then represents a set of concrete states, which is disjoint from the sets represented by other abstract states.

Abstraction is identified by a set of abstract states $\widehat{S}$, an abstraction mapping h, that associates with each concrete state the abstract state which represents it, and a set of atomic propositions AP which label the concrete and abstract states. We present three types of abstractions which differ in the choice of $\widehat{S}$, h, and AP: predicate abstraction, visible-variable abstraction, and data abstraction. We also suggest how an abstraction can be extracted from a high-level description of a program.

We describe the CounterExample-Guided Abstraction-Refinement (CEGAR) methodology which suggests an iterative, automated approach to verification with abstraction. We comment on different possible implementations for constructing the abstract model and its refinements.

2 Preliminaries

2.1 Temporal logics

Model checking algorithms typically use finite state transition systems to model the verified systems and propositional temporal logics to specify the desired properties. In this section we present the syntax and semantics of several subsets of the temporal logic CTL* [25].

Let AP be a set of atomic propositions. We define CTL* in *positive normal form*, in which negations are applied only to atomic propositions. This will facilitate the definition of universal and existential subsets of CTL* [29]. Since negations are not allowed, both conjunction and disjunction are required. Negations applied to the *next-time* operator $\mathbf{X}$ can be "pushed inwards" using the

logical equivalence $\neg(\mathbf{X}\, f) = \mathbf{X}\, \neg f$. The *unless* operator $\mathbf{V}$ (sometimes called the *release* operator), which is the dual of the *until* operator $\mathbf{U}$, is also added. Thus, $\neg(f\ \mathbf{U}\ g) = \neg f\ \mathbf{V}\ \neg g$.

Definition 1 (CTL*). *For a given set of atomic propositions AP, the logic CTL* is the set of state formulas, defined recursively by means of state formulas and path formulas as follows. State formulas are of the form:*

- *If $p \in AP$, then p and $\neg p$ are state formulas.*
- *If f and g are state formulas, then so are $f \wedge g$ and $f \vee g$.*
- *If f is a path formula, then $\mathbf{A}\, f$ and $\mathbf{E}\, f$ are state formulas.*

Path formulas are of the form:

- *If f is a state formula, then f is a path formula.*
- *If f and g are path formulas, then so are $f \wedge g$, and $f \vee g$.*
- *If f and g are path formulas, then so are $\mathbf{X}\, f$, $f\ \mathbf{U}\ g$, and $f\ \mathbf{V}\ g$.*

The abbreviations $true, false$ and implication $\rightarrow$ are defined as usual. For path formula f, the following abbreviations are widely used. $\mathbf{F}\, f \equiv true\ \mathbf{U}\ f$ express the properties that sometimes in the future *f holds. $\mathbf{G}\, f \equiv false\ \mathbf{V}\ f$ express the properties that f holds* globally.

CTL [14] is a branching-time subset of CTL* in which every temporal operator is immediately preceded by a path quantifier. Thus, nesting of temporal operators with no path quantifier in between is not allowed. Formally, CTL (in positive normal form) is the set of state formulas defined by:

- If $p \in AP$, then p and $\neg p$ are CTL formulas.
- If f and g are CTL formulas, then so are $f \wedge g$ and $f \vee g$.
- If f and g are CTL formulas, then so are $\mathbf{AX}\, f$, $\mathbf{A}(f\ \mathbf{U}\ g)$, $\mathbf{A}(f\ \mathbf{V}\ g)$ and $\mathbf{EX}\, f$, $\mathbf{E}(f\ \mathbf{U}\ g)$, $\mathbf{E}(f\ \mathbf{V}\ g)$.

ACTL* is the *universal* subset of CTL* in which only $\mathbf{A}$ path quantifiers are allowed. Similarly, ECTL* is the *existential* subset of CTL*in which only $\mathbf{E}$ path quantifiers are allowed. ACTL and ECTL are the restriction of ACTL* and ECTL* to CTL.

LTL [49] can be defined as the subset of ACTL* consisting of formulas of the form $\mathbf{A}\, f$, where f is a path formula in which the only state subformulas permitted are Boolean combinations of atomic propositions. More precisely, f is defined (in positive normal form) by

1. If $p \in AP$ then p and $\neg p$ are path formulas.
2. If f_1 and f_2 are path formulas, then $f_1 \wedge f_2$, $f_1 \vee f_2$, $\mathbf{X}\, f_1$, $f_1\, \mathbf{U}\, f_2$, and $f_1\, \mathbf{V}\, f_2$ are path formulas.

We will refer to such f as an LTL *path formula.*

The semantics of CTL* is defined with respect to a finite state transition system called *Kripke structure.*

Definition 2 (Kripke structure). *Let AP be a set of atomic propositions. A* Kripke structure *M over AP is a four-tuple $M = (S, S_0, R, L)$, where*

- *S is a (finite) set of states;*
- *$S_0 \subseteq S$ is the set of initial states;*
- *$R \subseteq S \times S$ is the transition relation, which must be* total*, i.e., for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$;*
- *$L : S \to \mathcal{P}(AP)$ is a function that labels each state with the set of atomic propositions true in that state.*

A *path* in M starting from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$, and for every $i \geq 0$, $R(s_i, s_{i+1})$. The suffix of π from state s_i is denoted π^i. The requirement that R is total simplifies the semantics of temporal logics over a Kripke structure since all paths are infinite. Several different semantics over finite paths can be found in [24].

We now consider the semantics of the logic CTL* with respect to a Kripke structure.

Definition 3 (Satisfaction of a formula). *Given a Kripke structure M, satisfaction of a state formula f by a model M at a state s, denoted $M, s \models f$, and of a path formula g by a path π, denoted $M, \pi \models g$, is defined as follows (where M is omitted when clear from the context).*

1. *$s \models p$ if and only if $p \in L(s)$; $s \models \neg p$ if and only if $p \notin L(s)$.*
2. *$s \models f \wedge g$ if and only if $s \models f$ and $s \models g$.*
   $s \models f \vee g$ if and only if $s \models f$ or $s \models g$.
3. *$s \models \mathbf{A}\, f$ if and only if for every path π from s, $\pi \models f$.*
   $s \models \mathbf{E}\, f$ if and only if there exists a path π from s such that $\pi \models f$.
4. *$\pi \models f$, where f is a state formula, if and only if the first state of π satisfies f.*
5. *$\pi \models f \wedge g$ if and only if $\pi \models f$ and $\pi \models g$.*
   $\pi \models f \vee g$ if and only if $\pi \models f$ or $\pi \models g$.
6. *(a) $\pi \models \mathbf{X}\, f$ if and only if $\pi^1 \models f$.*
   (b) $\pi \models f\ \mathbf{U}\ g$ if and only if for some $n \geq 0$, $\pi^n \models g$ and for all $i < n$, $\pi^i \models f$.
   (c) $\pi \models f\ \mathbf{V}\ g$ if and only if for all $n \geq 0$, if (for all $i < n$, $\pi^i \not\models f$) then $\pi^n \models g$.

$M \models f$ if and only if for every $s \in S_0$, $M, s \models f$.

In [25] it has been shown that CTL and LTL are incomparable in their expressive power, and that CTL* is more expressive than either of them.

2.2 Model Checking

Given a Kripke structure $M = (S, R, S_0, L)$ 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 whether the initial states are included in those states.

When M does not satisfy φ, model checking can provide a *counterexample* which demonstrates why the specification does not hold in the model. Counterexamples are very helpful for debugging. However, most model checking tools provide them only in limited cases. Common counterexamples have the form of either a finite path or a "lasso", which is a finite path followed by a simple cycle. The former is suitable for demonstrating why a specification of the form $\mathbf{AG}\, p$ fails to hold. It provides a finite path to a state satisfying $\neg p$. The latter is suitable to demonstrate why $\mathbf{AF}\, p$ fails. It shows an infinite path in a "lasso" shape along which all states satisfy $\neg p$. For general specifications, a tree or even a general graph is needed. Counterexamples for ACTL formulas are defined in [19] and for full CTL in [53].

Model checking has been successfully applied in hardware verification, and is emerging as an industrial standard tool for hardware design . A partial list of tools for hardware verification includes SMV [41] and NuSMV [12], FormalCheck [31], RuleBase [2], and Forecast [26]. Recently, several tools for model checking of software have been developed as well and applied to non-trivial examples. A partial list consists of SPIN [34], Bandera [21], Java PathFinder [32], SLAM, Bebop, and Zing [1], Blast [3], Magic [9], and CBMC [16]. An extensive overview of model checking algorithms can be found in [13].

The main technical challenge in model checking is the *state explosion* problem which occurs if the system is a composition of several components or if the system variables range over large domains.

An explicit state model checker is a program which performs model checking directly on a Kripke structure. SPIN [33] is an example of a successful tool of that kind. Large models are often handled implicitly. Two widely used approaches are the BDD-based [8, 42] and the SAT-based [4] model checking.

BDD-based model checking: Ordered Binary Decision Diagrams (BDDs) [7] are canonical representations of Boolean functions. They are often concise in their memory requirements. Furthermore, most operations needed for model checking can be defined in terms of Boolean functions and can be implemented efficiently with BDDs.

In BDD-based (also called *symbolic*) model checking, the transition relation of the Kripke structure is represented by a Boolean function, which in turn is represented by a BDD. Sets of states are also represented by Boolean functions. Fixpoint characterizations of temporal operators are applied to the Boolean functions representing the Kripke structure. BDDs are sometimes, but not always, exponentially smaller than explicit representation of the corresponding Boolean functions. In such cases, symbolic verification is successful.

Two operations are central to model checking. Given a set of states Q, *Image computation* computes the set of successors of states in Q:

$$Image(Q) := \{t \mid \exists s[R(s,t) \ \wedge \ Q(s)]\}.$$

Preimage computation computes the set of predecessors of states in Q:

$$Preimage(Q) := \{s \mid \exists t[R(s,t) \ \wedge \ Q(t)]\}.$$

Unfortunately, in contrast to pure Boolean operations, these operations are not efficiently computable [42], and their computation is a major bottleneck in symbolic model checking.

SAT-based model checking: Many problems, including some versions of model checking, can very naturally be translated into the *satisfiability* problem of propositional calculus. The satisfiability problem is known to be NP-complete. Nevertheless, modern SAT-solvers, developed in recent years, can handle formulas with several thousands of variables within a few seconds. SAT-solvers such as Grasp [39], Prover [52], Chaff [47], and Berkmin [27], and many others, are based on sophisticated learning techniques and data structures that accelerate the search for a satisfying assignment, if exists.

Below we describe a simple way to exploit satisfiability for bounded model checking of properties of the form $\mathbf{AG}\, p$, where p is a Boolean formula. *Bounded Model Checking* [5, 4] accepts a model M, a natural number (a bound) k, and a formula $\mathbf{AG}\, p$ as above. It constructs a propositional formula $f_{M,k}$, describing all computations of M of length k. It also constructs a propositional formula $f_{\varphi,k}$, describing all paths of length k satisfying the property $\varphi = \neg\, \mathbf{AG}\, p = \mathbf{EF}\, \neg p$. Next, it sends $f_{M,k} \wedge f_{\varphi,k}$ to a SAT-solver to check for satisfiability. If the formula is satisfiable then $M \not\models \mathbf{AG}\, p$ and the satisfying assignment corresponds to a computation of M, leading to a state satisfying $\neg p$. This path is a *counterexample* for the checked formula. If $f_{M,k} \wedge f_{\varphi,k}$ is unsatisfiable then no counterexample of length k exists in M. The bound k is then increased and the check is repeated.

The method described above is mainly suitable for refutation. Verification is obtained only if k exceeds the length of the longest path among all shortest paths from an initial state to some state in M. In practice, it is hard to compute this bound and even when known, it is often too large to handle. Full verification with SAT is possible using other methods, such as interpolation [43, 40], induction [51], and ALL-SAT [11, 44, 30]. However, these methods are more limited in their applicability to large systems. Bounded model checking can easily be extended for checking LTL formulas, interpreted over finite paths [5].

Many of the modern hardware verification tools such as NuSMV [12], RuleBase [2], Forecast & Thunder [26, 20], and FormalCheck [31] include both SAT and BDD methods and apply the one that is most successful in each case.

2.3 Equivalences and preorders

In this section we define relations on Kripke structures that guarantee logic preservation. The relations are *structural*. That is, they are defined by means of states and transitions of the Kripke structures. The structural relations correspond to *logical* relations that guarantee preservation of truth of formulas between related structures. These relations are exploited in many of the approaches

to avoiding the state explosion problem in model checking, such as, abstraction, modular model checking, symmetry, and partial-order reductions [13]. Instead of checking the full model of the system, a smaller model with guaranteed logic preservation is checked.

We define two structural relations: The *bisimulation relation* [48] and the *simulation preorder* [45]. Intuitively, two states are bisimilar if they are identically labeled and for every successor of one there is a bisimilar successor of the other. Similarly, one state is smaller than another by the simulation preorder if they are identically labeled and for every successor of the smaller state there is a corresponding successor of the greater one. The simulation preorder differs from bisimulation in that the greater state may have successors with no corresponding successors in the smaller state.

Let AP be a set of atomic propositions and let $M_1 = (S_1, S_{0_1}, R_1, L_1)$ and $M_2 = (S_2, S_{0_2}, R_2, L_2)$ be two structures over AP.

Definition 4. *A relation $B \subseteq S_1 \times S_2$ is a* bisimulation relation *[48] over M_1 and M_2 if the following conditions hold:*

1. *For every $s_1 \in S_{0_1}$ there is $s_2 \in S_{0_2}$ such that $B(s_1, s_2)$. Moreover, for every $s_2 \in S_{0_2}$ there is $s_1 \in S_{0_1}$ such that $B(s_1, s_2)$.*
2. *For every $(s_1, s_2) \in B$,*
   - *$L_1(s_1) = L_2(s_2)$ and*
   - *$\forall t_1[\ R_1(s_1, t_1) \longrightarrow \exists t_2[\ R_2(s_2, t_2) \ \wedge \ B(t_1, t_2)\ ]]$.*
   - *$\forall t_2[\ R_2(s_2, t_2) \longrightarrow \exists t_1[\ R_1(s_1, t_1) \ \wedge \ B(t_1, t_2)\ ]]$.*

We write $s_1 \cong s_2$ for $B(s_1, s_2)$. We say that M_1 and M_2 are *bisimilar* (denoted $M_1 \cong M_2$) if there exists a bisimulation relation B over M_1 and M_2.

Definition 5. *A relation $H \subseteq S_1 \times S_2$ is a* simulation relation *[46] over M_1 and M_2 if the following conditions hold:*

1. *For every $s_1 \in S_{0_1}$ there is $s_2 \in S_{0_2}$ such that $H(s_1, s_2)$.*
2. *For every $(s_1, s_2) \in H$,*
   - *$L_1(s_1) = L_2(s_2)$ and*
   - *$\forall t_1[\ R_1(s_1, t_1) \longrightarrow \exists t_2[\ R_2(s_2, t_2) \ \wedge \ H(t_1, t_2)\ ]]$.*

We write $s_1 \preceq s_2$ for $H(s_1, s_2)$. *M_2 simulates M_1* (denoted $M_1 \preceq M_2$) if there exists a simulation relation H over M_1 and M_2.

The following theorem relates bisimulation and simulation to the logics they preserve[1].

Theorem 1.

- *[6] Let $M_1 \cong M_2$. Then for every CTL* formula f (with atomic propositions in AP), $M_1 \models f$ if and only if $M_2 \models f$.*
- *[29] Let $M_1 \preceq M_2$. For every ACTL* formula f with atomic propositions in AP, $M_2 \models f$ implies $M_1 \models f$.*

[1] Bisimulation and simulation also preserve the μ-calculus logic [35] and its universal [37] subset, respectively. The discussion of μ-calculus is beyond the scope of this paper.

2.4 Programs and their models

We describe a simple syntactic framework to formalize programs. A *program* $\mathcal{P}$ has a finite set of variables $V = \{v_1, \cdots, v_n\}$ (sometimes also denoted as a tuple $\overline{v} = (v_1, \ldots, v_n)$), where each variable v_i has an associated domain D_{v_i}. The set of all possible states for program $\mathcal{P}$ is $D_{v_1} \times \cdots D_{v_n}$ which we denote by D. The value of a variable v in state s is denoted by $s(v)$. *Expressions* are built from variables in V, constants in D_{v_i}, and function symbols in the usual way, e.g. $v_1 + 3$. *Atomic formulas* are constructed from expressions and relation symbols, e.g. $v_1 + 3 < 5$. Similarly, *predicates* are composed of atomic formulas using negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$). Thus, predicates are in fact quantifier-free first order formulas. 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.

Predicates are used to identify initial states of the program as well as conditions in program statements such as **if** and **while**.

A specification for a program $\mathcal{P}$ is an ACTL* formula φ whose atomic formulas are predicates over the program variable. Let Atoms(φ) be the set of atomic formulas appearing in the specification φ. Atoms($\mathcal{P}$) is the set of atomic formulas that appear in the definition of initial states or in the conditions in the program.

Each program $\mathcal{P}$ naturally corresponds to a Kripke structure $M = (S, S_0, R, L)$, where $S = D$ is the set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and L is a labeling function given by $L(d) = \{f \in \text{Atoms}(\mathcal{P}) \mid d \models f\}$. Translating a program into a Kripke structure is straightforward and will not be described here.

3 Abstract models

In this section we define an abstract model (Kripke structure) based on a given concrete one. The abstract model is guaranteed by construction to be greater than the concrete model by the simulation relation, thus preservation of universal logics is obtained. In practice, however, the concrete model is too large to fit into memory and therefore is never produced. The abstract models are in fact constructed directly from some high-level description of the system.

For simplicity we consider abstractions obtained by collapsing disjoint sets of concrete states (in S) into single abstract states (in $\widehat{S}$). We will not consider here non-disjoint sets, as is done for instance in *Abstract Interpretation* [37, 22].

We use a function $h : S \to \widehat{S}$, called the *abstraction mapping*, to map each concrete state to the abstract state that represents it. The abstraction mapping h induces an equivalence relation $\equiv_h$ on the domain S in the following manner: Let s, t be states in S, then

$$s \equiv_h t \text{ iff } h(s) = h(t).$$

Since an abstraction can be represented either by an abstraction mapping 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 $\equiv_h$.

3.1 Existential Abstraction

We define abstract Kripke structures by means of *existential abstraction* [15]. Existential abstraction defines an abstract state to be an initial state if it represents an initial concrete state. Similarly, there is a transition from abstract states $\widehat{s}$ to abstract state $\widehat{s'}$ if there is a transition from a state represented by $\widehat{s}$ to a state represented by $\widehat{s'}$ (see Figure 1). Formally,

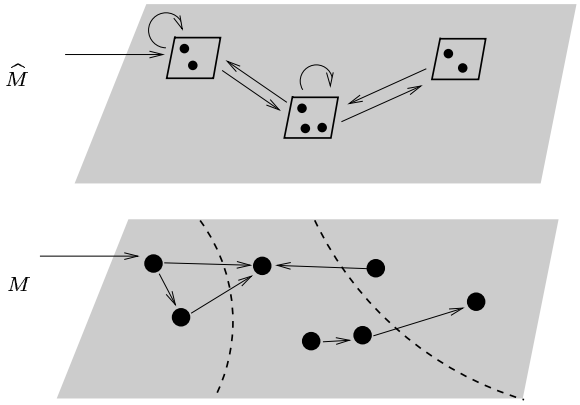

Fig. 1. Existential Abstraction. M is the original Kripke structure, and $\widehat{M}$ the abstracted one. The dotted lines in M indicate how the states of M are clustered into abstract states.

Definition 6. *Let $M = (S, S_0, R, L)$ be a (concrete) Kripke structure, let $\widehat{S}$ be a set of abstract states and $h : S \to \widehat{S}$ be an abstraction mapping. The* abstract Kripke structure *$\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L})$ generated by h for M is defined as follows:*

1. $\widehat{S_0}(\widehat{s})$ *iff* $\exists s\,(h(s) = \widehat{s} \ \wedge \ S_0(s))$.
2. $\widehat{R}(\widehat{s_1}, \widehat{s_2})$ *iff* $\exists s_1 \exists s_2\,(h(s_1) = \widehat{s_1} \ \wedge \ h(s_2) = \widehat{s_2} \ \wedge \ R(s_1, s_2))$.
3. $\widehat{L}(\widehat{s}) = \bigcap_{h(s)=\widehat{s}} L(s)$.

Having 'iff' in items 1 and 2 of the definition above results in the *exact* abstract model of M, with respect to h. Replacing 'iff' by 'if' results in a model with more initial states and more transitions, which still over-approximates the structure M. Such a model is sometimes easier to construct. The results below hold for any abstract Kripke structure constructed by existential abstraction, not only for the exact one.

Note that, $\widehat{s}$ is labeled by an atomic proposition if and only if all the states it represents are labeled by that proposition. We would like the abstract model to satisfy as many atomic propositions as possible. In order to achieve this, we introduce a condition on the abstraction mapping, guaranteeing that all concrete states in an equivalence class of $\equiv_h$ share the same labels.

An abstraction mapping h is *appropriate* for a specification φ if for all atomic formulas $f \in Atoms(\varphi)$, and for all states s and t in S such that $s \equiv_h t$ it holds that $s \models f \Leftrightarrow t \models f$.

Let M and φ be defined over AP and let h be appropriate for φ, then $s \equiv_h t$ implies $L(s) = L(t)$. Moreover, $h(s) = \widehat{s}$ implies $\widehat{L}(\widehat{s}) = L(s)$.

The following theorem shows that for ACTL*, specifications which are correct for $\widehat{M}$ are correct for M as well.

Theorem 2. *Let M be a Kripke structure and φ be an ACTL* formula, both defined over AP. Further, let h be appropriate for φ. Then $M \preceq \widehat{M}$. Consequently, $\widehat{M} \models \varphi$ implies $M \models \varphi$.*

Note that once $\widehat{S}$, h, and AP are given, $\widehat{S_0}$, $\widehat{R}$, and $\widehat{L}$ are uniquely determined. Thus, $\widehat{S}$, h, and AP uniquely determine $\widehat{M}$. Since h implicitly includes the information about $\widehat{S}$ and AP, we sometimes refer to h for identifying $\widehat{M}$. In the next subsections we will define different types of abstraction by means of their abstract states and abstraction mapping. Other abstraction types are also possible.

3.2 Abstraction Types

Let $\mathcal{P}$ be a program and let φ be an ACTL* formula. We describe several ways to define abstractions that are appropriate for checking φ on $\mathcal{P}$. They are all based on the existential abstraction. They differ from each other in their choice of abstract states, in the set of atomic propositions that label both concrete and abstract states and in the definition of the abstraction function h.

Predicate Abstraction: Predicate abstraction [28, 50] is based on a set of predicates $\{P_1, \ldots, P_k\}$, defined over the program variables. Recall that predicates are quantifier-free first order formulas. Since our goal is to check a property φ on a program $\mathcal{P}$, $Atoms(\varphi)$, the set of predicates appearing in φ, must be included in the set of predicates. In addition, this set will contain some of the conditions in control statements in $\mathcal{P}$, and possibly other predicates.

In order to define the abstract state space, each predicate P_j is associated with a Boolean variable B_j. The set of abstract states are valuations of $\{B_1, \ldots, B_k\}$. Thus, $\widehat{S} = \{0,1\}^k$.

The predicates are also used to define the abstraction mapping h between the concrete and abstract state spaces. A concrete state s will be mapped to an abstract state $\widehat{s}$ through h if and only if the truth value of each predicate on s

equals the value of the corresponding Boolean variable in the abstract state $\widehat{s}$. Formally,

$$h(s) = \widehat{s} \quad \Leftrightarrow \bigwedge_{1 \leq j \leq k} (P_j(s) \Leftrightarrow B_j(\widehat{s})). \tag{1}$$

The predicates also serve as the atomic propositions that label the states in the concrete and abstract models. That is, the set of atomic propositions is $AP = \{P_1, P_2, .., P_k\}$. A state in the concrete system will be labeled with all the predicates it satisfies. Note that, all concrete states mapped to the same abstract state $\widehat{s}$ agree on the values of all predicates P_j and also agree with $\widehat{s}$ on the value of the corresponding B_j. Thus, an abstract state will be labeled with predicate P_j if and only if the corresponding bit B_j is 1 in that state.

Note also that h is a function because each P_j can have one and only one value on a given concrete state and so the abstract state corresponding to the concrete state is unique. h is also appropriate for any ACTL* formula over AP, and in particular φ.

Once $\widehat{S}$, h, and AP have been determined, the rest of the abstract model is defined as explained before, by means of existential abstraction.

Example 1. We will exemplify some of the notions defined above on a simple example. Consider a program $\mathcal{P}$ with variables x, y over the natural numbers and a single transition $x := x + 1$. Let $AP = \{P_1, P_2, P_3\}$ where $P_1 = (x \leq 1)$, $P_2 = (x > y)$, and $P_3 = (y = 2)$.

Let s and t be two concrete states such that $s(x) = s(y) = 0$, $t(x) = 1$ and $t(y) = 2$. Then, $L(s) = \{P_1\}$ and $L(t) = \{P_1, P_3\}$.

The abstract states are defined over valuations of the Boolean variable B_1, B_2, B_3. Thus, $\widehat{S} \subseteq \{0,1\}^3$. The abstraction mapping h is: $h(s) = (1,0,0)$ and $h(t) = (1,0,1)$. Note that $\widehat{L}((1,0,0)) = L(s) = \{P_1\}$, where $\widehat{L}((1,0,1)) = L(t) = \{P_1, P_3\}$. The abstract transition relation can be represented by the following formula:

$$\begin{array}{l} \widehat{R}(B_1, B_2, B_3, B_1', B_2', B_3') \Longleftrightarrow \\ \exists x, y, x', y' [\ P_1(x,y) \Leftrightarrow B_1 \ \wedge P_2(x,y) \Leftrightarrow B_2 \ \wedge \ P_3(x,y) \Leftrightarrow B_3 \ \wedge \\ x' = x + 1 \ \wedge \ y' = y \ \wedge \\ P_1(x',y') \Leftrightarrow B_1' \ \wedge \ P_2(x',y') \Leftrightarrow B_2' \ \wedge \ P_3(x',y') \Leftrightarrow B_3' \ ]. \end{array}$$

If the program $\mathcal{P}$ is over a finite, relatively small state space, then BDDs can be used to compute $\widehat{R}$. For that, we will need a BDD representation of the concrete transition relation R (possibly in the form of a partitioned transition relation [13]). Further, we will need a BDD representation for h.

If the program is over a finite but large state space then SAT solvers will be more appropriate, while if its state space is infinite then theorem prover will have to be used [50].

The two other types of abstractions described next can both be defined by means of predicate abstraction. However, they are interesting special cases.

Abstraction based on visible and invisible variables: The visible-variables abstraction, also known as *localization reduction* [36], is based on a partition of the program variables into visible and invisible variables. It is a simpler special case of predicate abstraction and is widely used in model checking of hardware. The visible variables, denoted $\mathcal{V}$, are considered to be important for the checked property φ and hence are retained in the abstract model. This set includes, in particular, all variables that appear in φ. The rest of the variables, called *invisible*, are considered irrelevant for checking φ. Ideally, only a small subset of the variables will be considered visible.

Formally, given a set of variables $U = \{u_1, \ldots, u_p\}$, $U \subseteq V$, let s^U denotes the portion of s that corresponds to variables in U, i.e., $s^U = (s(u_1), \ldots, s(u_p))$

Let $\mathcal{V} = \{u_1, \ldots, u_q\} \subseteq V$ be the set of visible variables. Then, the set of abstract states is $\widehat{S} = D_{u_1} \times \ldots \times D_{u_q}$. The abstraction function $h : S \to \widehat{S}$ is defined as $h(s) = s^{\mathcal{V}}$. AP includes all atomic propositions in φ. Since all variables that appear in φ are visible, h is appropriate for φ.

A conservative choice of the set of visible variables is described below. Assume that each variable $v \in V$ is associated with a next-state function $f_v(V)$. Typically, f_v depends only on a subset of V. The Cone Of Influence (COI) [13] of a formula φ is defined inductively as follows. It includes all the variables in φ. In Addition, if v is in COI, then all variables on which f_v depends are also in COI.

Taking the COI of φ to be the set of visible variable, results in an abstract model which is *equivalent* to the concrete model with respect to φ. That is, the abstract model satisfies φ if and only if the concrete model satisfies it. As a result, refutation of φ on the abstract model implies refutation on the concrete model. This choice, however, is often not practical, since COI is typically too large.

Note that, in contrast to predicate abstraction, the visible-variables abstraction cannot retain any information on variables over infinite domain. Such variables must be considered invisible. This is because the domain of a visible variable is taken as is and no abstraction is applied to it. In the next section we present an abstraction that can abstract domains of individual variables.

Data abstraction: Another useful abstraction can be obtained by abstracting away some of the data information. *Data abstraction* [15, 38] is done by choosing, for every variable in the system, an abstract domain that is typically significantly smaller than the original domain. The abstraction function maps concrete data domains to abstract data domains and induces an abstraction function from concrete states to abstract states.

Clearly, a property verified for the abstract model can only refer to the abstract values of the program variables. In order for such a property to be meaningful in the concrete model we label the concrete states by atomic formulas of the form $\widehat{v_i} = a$, where a is an element of the abstract domain of v_i. These atomic formulas indicate that the variable v_i has some value d that has been abstracted to a.

Let $\mathcal{P}$ be a program with variables $v_1, \ldots, v_n$. For simplicity we assume that all variables are over the same domain D. Thus, the concrete model of the system is defined over states s of the form $s = (d_1, \ldots, d_n)$ in $D \times \ldots \times D$, where d_i is the value of v_i in this state.

In order to build an abstract model for $\mathcal{P}$ we need to choose an abstract domain A and an variable-abstraction mapping $h : D \to A$. The abstract state space is then defined by

$$\widehat{S} = A \times \ldots \times A.$$

The abstraction mapping is an extension of the variable-abstraction mapping h to n-tuples in $D \times \ldots \times D$. By abuse of notion we denote the abstraction mapping by h as well.

$$h((d_1, \ldots, d_n)) = (h(d_1), \ldots, h(d_n)).$$

As before, an abstract state $(a_1, \ldots, a_n)$ of $\widehat{S}$ represents the set of all states $(d_1, \ldots, d_n)$ such that $h((d_1, \ldots, d_n)) = (a_1, \ldots, a_n)$.

The next step is to restrict the concrete model of $\mathcal{P}$ so that it reflects only the abstract values of its variables. This is done by defining the set of atomic propositions as follows:

$$AP = \{\ \widehat{v_i} = a \mid i = 1, \ldots, n \ \text{ and } \ a \in A\ \}.$$

The notation $\widehat{v_i}$ is used to emphasize that we refer to the abstract value of the variable v_i. The labeling of a state $s = (d_1, \ldots, d_n)$ in the concrete model will be defined by

$$L(s) = \{\ \widehat{v_i} = a_i \mid h(d_i) = a_i,\ i = 1, \ldots, n\ \}.$$

By restricting the state labeling we lose the ability to refer to the actual values of the program variables. However, many of the states are now indistinguishable and can be collapsed into a single abstract state.

Here again all states mapped to an abstract state agree on all atomic propositions. Thus, the abstraction mapping h is appropriate for every ACTL* formula defined over AP. The abstract labeling, defined according to the existential abstraction, can also be described as follows. Let $\widehat{s} = (a_1, \ldots, a_n)$. Then,

$$\widehat{L}(\widehat{s}) = \{\ \widehat{v_i} = a_i \mid i = 1, \ldots, n\ \}.$$

Example 2. Let $\mathcal{P}$ be a program with a variable x over the integers. Let s, s' be two program states such that $s(x) = 2$ and $s'(x) = -7$. Following are two possible abstractions.

Abstraction 1:

$$A_1 = \{a_-, a_0, a_+\} \text{ and}$$

$$h_1(d) = \begin{cases} a_+ \text{ if } d > 0 \\ a_0 \text{ if } d = 0 \\ a_- \text{ if } d < 0 \end{cases}$$

Thus, $h(s) = (a_+)$ and $h(s') = (a_-)$. The set of atomic propositions is $AP_1 = \{\ \widehat{x} = a_-,\ \widehat{x} = a_0,\ \widehat{x} = a_+\ \}$.
The labeling of states in the concrete and abstract models induced by A_1 and h_1 is:
$L_1(s) = \widehat{L}(a_+) = \{\widehat{x} = a_+\}$ and $L_1(s') = \widehat{L}(a_-) = \{\widehat{x} = a_-\}$.

Abstraction 2:

$$A_2 = \{a_{even},\ a_{odd}\} \text{ and}$$

$$h_2(d) = \begin{cases} a_{even} \text{ if } even(|d|) \\ a_{odd} \text{ if } odd(|d|) \end{cases}$$

Here $h(s) = (a_{even})$ and $h(s') = (a_{odd})$. The set of atomic propositions is $AP_2 = \{\ \widehat{x} = a_{even},\ \widehat{x} = a_{odd}\ \}$.
The labeling induced by A_2 and h_2 is:
$L_2(s) = \widehat{L}(a_{even}) = \{\widehat{x} = a_{even}\}$ and $L_2(s') = \widehat{L}(a_{odd}) = \{\widehat{x} = a_{odd}\}$.

4 Deriving models from the program text

In the next section we explain how the exact and approximated abstract model for the system can be derived directly from a high-level description of a program. In order to avoid having to choose a specific programming language, we argue that the program can be described by means of first-order formulas. In this section we demonstrate how this can be done.

Let $\mathcal{P}$ be a program, and let $\overline{v} = (v_1, \ldots, v_n)$ and $\overline{v}' = (v'_1, \ldots, v'_n)$ be two copies of the program variables, representing the current and next state, respectively. The program will be given by two first-order formulas, $\mathcal{S}_0(\overline{v})$ and $\mathcal{R}(\overline{v}, \overline{v}')$, describing the set of initial states and the set of transitions, respectively. Let $\overline{d} = (d_1, \ldots, d_n)$ be a vector of values. The notation $\mathcal{S}_0[\overline{v} \leftarrow \overline{d}]$ indicates that for every $i = 1, \ldots, n$, the value d_i is substituted for the variable v_i in the formula $\mathcal{S}_0$. A similar notation is used for substitution in the formula $\mathcal{R}$.

Definition 7. *Let $S = D \times \ldots \times D$ be the set of states in a model M. The formulas $\mathcal{S}_0(\overline{v})$ and $\mathcal{R}(\overline{v}, \overline{v}')$ define the set of initial states S_0 and the set of transitions R in M as follows. Let $s = (d_1, \ldots, d_n)$ and $s' = (d'_1, \ldots, d'_n)$ be two states in S.*

- $S_0(s) \Leftrightarrow \mathcal{S}_0(\overline{v})[\overline{v} \leftarrow \overline{d}]$ *is true.*
- $R(s, s') \Leftrightarrow \mathcal{R}(\overline{v}, \overline{v}')[\overline{v} \leftarrow \overline{d}, \overline{v'} \leftarrow \overline{d'}]$ *is true.*

The following example demonstrates how a program can be described by means of first-order formulas. A more elaborate explanation can be found in [13]. We assume that each statement in the program starts and ends with labels that uniquely define the corresponding locations in the program. The program locations will be represented in the formula by the variable *pc* (the *program counter*), which ranges over the set of program labels.

Example 3. Given a program with one variable x that starts at label l_0, in any state in which x is even, the set of its initial states is described by the formula:

$$\mathcal{S}_0(pc, x) = \quad pc = l_0 \ \wedge \ even(x).$$

Let $l: \ x := e \ \ l'$ be a program statement. the transition relation associated with this statement is described by the formula:

$$\mathcal{R}(pc, x, pc', x') = \ pc = l \ \wedge \ x' = e \ \wedge \ pc' = l'.$$

Given the statement $l: \ \text{if } x = 0 \text{ then } l_1: \ x := 1 \text{ else } l_2: \ x := x + 1 \ l'$, the transition relation associated with it is described by the formula:

$$\begin{aligned}\mathcal{R}(pc, x, pc', x') = (&(\ pc = l \ \wedge \ x = 0 \ \wedge x' = x \wedge pc' = l_1) \vee \\ &(\ pc = l \ \wedge \ x \neq 0 \ \wedge x' = x \wedge pc' = l_2) \vee \\ &(pc = l_1 \wedge x' = 1 \wedge pc' = l') \vee \\ &(pc = l_2 \wedge x' = x + 1 \wedge pc' = l')).\end{aligned}$$

Note that checking the condition of the if statement takes one transition, along which the value of the program variable is checked but not changed. If the program contains an additional variable y, then $y' = y$ will be added to the description of each of the transitions above. This captures the fact that variables that are not assigned a new value keep their previous value.

4.1 Deriving abstract models

Let $\mathcal{S}_0$ and $\mathcal{R}$ be the formulas describing a concrete model M. Let $\widehat{S}$ and h be the set of abstract states and the abstraction mapping, defining an abstract model $\widehat{M}$. We would like to define formulas $\widehat{\mathcal{S}}_0$ and $\widehat{\mathcal{R}}$ that describe the model $\widehat{M}$. We first define formulas that describe the *exact* abstract model. To emphasize that it is the exact model we will denote it by $\widehat{M}_e$. We then show how to construct formulas describing an *approximated* abstract model with possibly more initial states and more transitions. The latter formulas represent an abstract model which is less precise than the exact one, but is easier to compute.

Let $\widehat{S}$ be defined over the variables $(\widehat{v_1}, \ldots, \widehat{v_k})$, that is $\widehat{S}$ is the set of valuations of those variables. Further, let $h((v_1, \ldots, v_n))) = (\widehat{v_1}, \ldots, \widehat{v_k})$. Thus, h maps valuations over the concrete domain to valuations over the abstract domain. The new formulas that we construct for $\widehat{M}$ will be defined over the variables $(\widehat{v_1}, \ldots, \widehat{v_k})$. The formulas will determine for abstract states whether they are initial and whether there is a transition connecting them. For this purpose, we first define a derivation of a formula over variables $\widehat{v_1}, \ldots, \widehat{v_k}$ from a formula over $v_1, \ldots, v_n$.

Definition 8. *Let φ be a first-order formula over variables $v_1, \ldots, v_n$. The formula $[\varphi]$ over $\widehat{v_1}, \ldots, \widehat{v_k}$ is defined as follows:*

$$[\varphi](\widehat{v_1}, \ldots, \widehat{v_k}) = \exists v_1 \ldots v_n \left(h((v_1, \ldots, v_n))) = (\widehat{v_1}, \ldots, \widehat{v_k}) \wedge \varphi(v_1, \ldots, v_n)\right).$$

Lemma 1. *Let $\mathcal{S}_0$ and $\mathcal{R}$ be the formulas describing a model M. Then the formulas $\widehat{\mathcal{S}}_0 = [\mathcal{S}_0]$ and $\widehat{\mathcal{R}} = [\mathcal{R}]$ describe the exact model $\widehat{M}_e$.*

The lemma holds since $\widehat{M}_e$ is defined by existential abstraction (see Definition 6). This is directly reflected in $[\mathcal{S}_0]$ and $[\mathcal{R}]$.

Using $\widehat{\mathcal{S}}_0$ and $\widehat{\mathcal{R}}$ allows us to build the exact model $\widehat{M}_e$ without first building the concrete model M. However, the formulas $\mathcal{S}_0$ and $\mathcal{R}$ might be quite large. Thus, applying existential quantification to them might be computationally expensive. We therefore define a transformation $\mathcal{T}$ on first-order formulas. The idea of $\mathcal{T}$ is to push the existential quantification inwards, so that it is applied to simpler formulas.

Definition 9. *Let φ be a first-order formula in positive normal form. Then $\mathcal{T}(\varphi)$ is defined as follows:*

1. *If p is a predicate, then $\mathcal{T}(p(v_1, \ldots, v_n)) = [p](\widehat{v}_1, \ldots, \widehat{v}_k)$ and $\mathcal{T}(\neg p(v_1, \ldots, v_n)) = [\neg p](\widehat{v}_1, \ldots, \widehat{v}_k)$.*
2. *$\mathcal{T}(\varphi_1 \wedge \varphi_2) = \mathcal{T}(\varphi_1) \wedge \mathcal{T}(\varphi_2)$.*
3. *$\mathcal{T}(\varphi_1 \vee \varphi_2) = \mathcal{T}(\varphi_1) \vee \mathcal{T}(\varphi_2)$.*
4. *$\mathcal{T}(\forall v \varphi) = \forall \widehat{v} \mathcal{T}(\varphi)$.*
5. *$\mathcal{T}(\exists v \varphi) = \exists \widehat{v} \mathcal{T}(\varphi)$.*

We can now define an *approximated* abstract model $\widehat{M}$. It is defined over the same set of states as the exact model, but its set of initial states and set of transitions are defined using the formulas $\mathcal{T}(\mathcal{S}_0)$ and $\mathcal{T}(\mathcal{R})$. The following lemma ensures that every initial state of $\widehat{M}_e$ is also an initial state of $\widehat{M}$. Moreover, every transition of $\widehat{M}_e$ is also a transition of $\widehat{M}$.

Lemma 2. *For every first-order formula φ in positive normal form, $[\varphi]$ implies $\mathcal{T}(\varphi)$. In particular, $[\mathcal{S}_0]$ implies $\mathcal{T}(\mathcal{S}_0)$ and $[\mathcal{R}]$ implies $\mathcal{T}(\mathcal{R})$.*

Note that the other direction does not hold. Cases 2 and 4 of Definition 9 result in nonequivalent formulas.

Corollary 1. *$M \preceq \widehat{M}_e \preceq \widehat{M}$.*

By allowing $\widehat{M}$ to have more behaviors than $\widehat{M}_e$, we increase the likelihood that it will falsify ACTL* formulas that are actually true in the concrete model and possibly true in $\widehat{M}_e$. This reflects the tradeoff between the precision of the model and the ease of its computation.

In practice, there is no need to construct formulas in order to build the approximated model. Let p be a predicate associated with a basic action a in the program (e.g. conditions, assignments of mathematical expressions). The user should provide *abstract predicates* $[p]$ and $[\neg p]$ for every such action a. Based on these, the approximated model can be constructed automatically.

In [15, 38], the construction of abstract models presented here has been developed and applied in the context of data abstraction.

5 Counterexample-guided Abstraction Refinement

It is easy to see that, regardless of the type of abstraction we use, the abstract model $\widehat{M}$ contains less information than the concrete model M[2]. Thus, model checking the structure $\widehat{M}$ potentially produces incorrect results. Theorem 2 guarantees that if an ACTL* specification is true in $\widehat{M}$ then it is also true in M. 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 4. The US traffic light controller presented in Figure 2, is defined over atomic propositions $AP = \{state = red\}$. We would like to prove for it the formula $\psi = \mathbf{AG\,AF}(state = red)$ using the abstraction mapping $h(red) = \widehat{red}$ 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 $\langle \widehat{red}, \widehat{go}, \widehat{go}, \ldots \rangle$ that invalidates the specification. However no corresponding concrete trace exists.

Fig. 2. Abstraction of a US traffic light.

When an abstract counterexample does not correspond to some concrete counterexample, we call it *spurious.* For example, $\langle \widehat{red}, \widehat{go}, \widehat{go}, \ldots \rangle$ 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 go, we will end up in state D, from which we cannot go 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. Finally, state I is an *irrelevant state* since it is neither deadend nor bad. To eliminate the spurious path, the abstraction can be refined, for instance, as indicated by the thick line, separating deadend states from bad states.

5.1 The Abstraction-Refinement Framework for ACTL*

In this section we present the framework of *CounterExample-Guided Abstraction-Refinement* (CEGAR), for the logic ACTL* and existential abstraction. The main steps of the CEGAR framework are presented below:

[2] From now on we will assume that $\widehat{M}$ is defined according to an abstraction mapping h which is appropriate for the checked property.

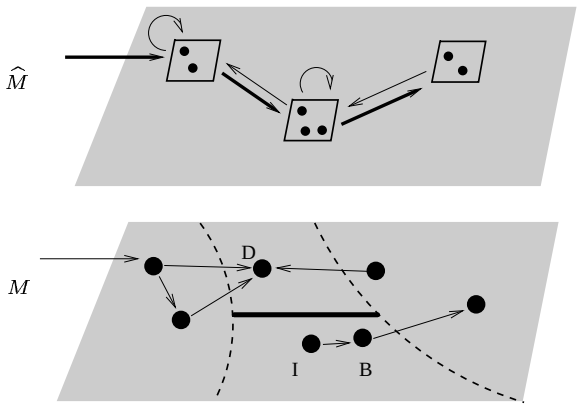

Fig. 3. The abstract path in $\widehat{M}$ (indicated by the thick arrows) is spurious. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line in M.

1. Given a model M and an ACTL* formula φ, generate an initial abstract model $\widehat{M}$.
2. Model check $\widehat{M}$ with respect to φ [3]. If φ is true, then conclude that the concrete model satisfies the formula and stop. If a counterexample $\widehat{T}$ is found, check whether it is also a counterexample in the concrete model. If it is, conclude that the concrete model does not satisfy the formula and stop. Otherwise, the counterexample is spurious. Proceed to step 3.
3. Refine the abstract model, so that $\widehat{T}$ will not be included in the new, refined abstract model. Go back to step 2.

Suggesting an initial abstraction and refinements manually requires great ingenuity and close acquaintance with the verified system. Here we present a framework, developed in [17], in which both steps are done automatically. The initial abstraction is constructed based on the program text, and refinements are determined by spurious counterexamples.

5.2 Detailed Overview of CEGAR

We now describe in more detail the CEGAR framework for ACTL*. For a program $\mathcal{P}$ and an ACTL* formula φ, our goal is to check whether the Kripke structure M corresponding to $\mathcal{P}$ satisfies φ. The CEGAR methodology consists of the following steps.

1. *Generate the initial abstraction:* We generate an initial abstraction mapping h by examining the program text. We consider the conditions used in control statements such as **if**, **while**, and **case**, and also the atomic formulas in φ.

[3] Most existing model checking tools handle CTL or LTL which are subsets of ACTL*.

The initial abstraction is an existential abstraction, constructed according to one of the abstractions described in Section 3.2.

2. *Model-check the abstract structure:* Let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction mapping h. We check whether $\widehat{M} \models \varphi$. If the check is affirmative, then we can conclude that $M \models \varphi$ (see Theorem 2). Suppose the check reveals that there is a counterexample $\widehat{T}$. We ascertain whether $\widehat{T}$ is an actual counterexample, i.e., it corresponds to a counterexample in the unabstracted structure M. If $\widehat{T}$ turns out to be an actual counterexample, we report it to the user, otherwise $\widehat{T}$ is a spurious counterexample, and we proceed to step 3.
3. *Refine the abstraction:* We refine the abstraction mapping h by partitioning a *single equivalence class* of $\equiv$ so that after the refinement, the refined abstract structure $\widehat{M}$ does not admit the spurious counterexample $\widehat{T}$. We will not discuss here partitioning algorithms. After refining the abstraction mapping, we return to step 2.
   The refinement can be accelerated in the cost of faster increase of the abstract model if the criterion obtained for partitioning one equivalence class (e.g. a new predicate) is used to partition all classes.

Depending on the type of h and the size of M, the initial abstract model (i.e., abstract initial states and abstract transitions) can be built using BDDs, SAT solvers or theorem provers. Similarly, the partitioning of abstract states, performed in the refinement, can be done using BDDs (e.g. as in [17]), SAT solvers (e.g. as in [10]), or linear programming and machine learning (e.g. as in [18]).

5.3 BDD-based CEGAR

In this section we describe a BDD-based implementation of the CEGAR framework, in which the initial abstraction and the refinements are computed and represented symbolically, using BDDs.

Model Checking the Abstract Model We use standard symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the specification φ. If it does, then by Theorem 2 we can conclude that the original Kripke structure also satisfies φ. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the rest of this section, we will focus on counterexamples which are *finite paths*. In [17], counterexamples consisting of a finite path followed by a loop are also considered. In [19], tree-like counterexamples for all of ACTL are considered.

5.4 Identification of Spurious Path Counterexamples

Assume the counterexample $\widehat{T}$ is a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. Given an abstract state $\widehat{s}$, the set of concrete states s such that $h(s) = \widehat{s}$ is denoted by $h^{-1}(\widehat{s})$, i.e.,

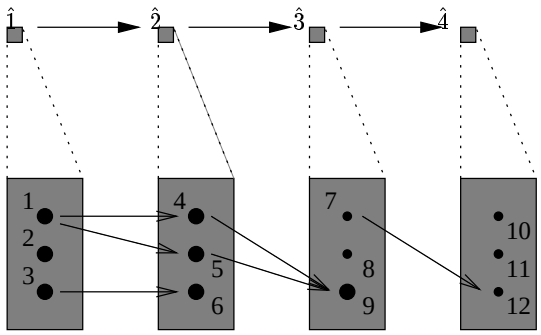

Fig. 4. An abstract counterexample

$h^{-1}(\widehat{s}) = \{s | h(s) = \widehat{s}\}$. We extend h^{-1} to sequences in the following way: $h^{-1}(\widehat{T})$ is the set of concrete paths defined as follows:

$$h^{-1}(\widehat{T}) = \{\langle s_1, \cdots, s_n\rangle | \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \ \wedge \ S_0(s_1) \ \wedge \ \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1})\}.$$

Next, we give a *symbolic* algorithm to compute $h^{-1}(\widehat{T})$. Let $S_1 = h^{-1}(\widehat{s_1}) \cap S_0$. For $1 < i \leq n$, we define S_i in the following manner: $S_i := Image(S_{i-1}) \cap h^{-1}(\widehat{s_i})$. Recall that, $Image(S_{i-1})$ is the set of successors of states in S_{i-1}. The sequence of sets S_i is computed symbolically using BDDs and the standard image computation algorithm. The following lemma establishes the correctness of this procedure.

Lemma 3. *The following are equivalent:*

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

Suppose that condition (iii) of Lemma 3 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 $\widehat{T}$. It follows from Lemma 3 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 $S_i = \emptyset$.

Example 5. In this example we apply data abstraction. Consider a program with only one variable with domain $D = \{1, \cdots, 12\}$. Assume that the abstraction mapping h maps $d \in D$ to $\lfloor (d-1)/3 \rfloor + 1$. 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 4; small dots denote non-reachable states. Suppose that we obtain an abstract counterexample

$\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 3, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that S_4 is empty. Thus, $\widehat{s_3}$ is the failure state.

Algorithm SplitPATH($\widehat{T}$)

$S := h^{-1}(\widehat{s_1}) \cap S_0$
$j := 1$
while ($S \neq \emptyset$ and $j < n$) {
    $j := j + 1$
    $S_{\text{prev}} := S$
    $S := Image(S) \cap h^{-1}(\widehat{s_j})$ }
if $S \neq \emptyset$ **then** output "counterexample exists"
else output j-1, S_{prev}

Fig. 5. SplitPATH checks if an abstract path is spurious.

The symbolic Algorithm **SplitPATH** in Figure 5 computes the index of the failure state 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. On the other hand, if the conditions stated in Lemma 3 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

5.5 Refining the Abstraction

In this section we explain how to refine an abstraction to eliminate the spurious counterexample. In order to simplify the discussion we assume that the abstract model is exact (see the discussion following Definition 6). Less precise abstract models can also be handled. Recall the discussion concerning Figure 3 in Section 5.1 where we identified deadend states, bad states, and irrelevant states. The refinement should suggest a partitioning of equivalence classes, that will separate the deadend states S_D from the bad states S_B.

We already have the deadend states. S_D is exactly the set S_{prev}, returned by the algorithm **SplitPATH**($\widehat{T}$). The algorithm also returns $j - 1$, the index in the counterexample where the failure state has been encountered. We can now compute the bad states symbolically as follows:

$$S_B = PreImage(h^{-1}(\widehat{s_{j+1}})) \cap h^{-1}(\widehat{s_j}).$$

$h^{-1}(\widehat{s_j})$ should now be partitioned to separate these two sets of states. This can be done in different ways. For example, if we work directly with BDDs, then we can add a new abstract state $\widehat{s_j'}$ to $\widehat{S}$ and update the BDD for h so that states

in S_D are now mapped to the new state $\widehat{s'_j}$. Of course, now $\widehat{R}$, $\widehat{S_0}$ and $\widehat{L}$ should be updated.

Our refinement procedure continues to refine the abstraction mapping by partitioning equivalence classes until a real counterexample is found, or the property is verified. If the concrete model is finite, then the partitioning procedure is guaranteed to terminate.

6 Conclusion

We surveyed abstractions based on over-approximation and preserving truth results of universal branching-time logics from the abstract model to the concrete model.

We did not cover many other approaches to abstraction. Those are usually based on more elaborated models. Such models allow, for instance, for abstract states to represent non-disjoint sets of concrete states. Others allow two types of transitions that over- or under-approximate the concrete transition relation and thus preserve the truth of full branching-time logics. Others allow to interpret formulas over 3-valued semantics, and can preserve both truth and falsity of full branching-time logics.

A relevant question for the abstraction-refinement framework is whether every infinite-state model has a finite-state abstraction (sometimes referred to as completeness). This question has been discussed for branching-time logics in, e.g., [23]. It turns out that some notion of fairness is needed in order to guarantee completeness. It should be noted that for a finite model this question does not arise since it can always serve as its own abstraction. It should also be noted that even for complete abstraction the iterative process of abstraction-refinement is not guaranteed to terminate since there is no constructive way to construct the abstraction.

References

1. T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in VErification (WAVE)*, July 2000.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: an industry-oriented formal verification tool. In *proceedings of the 33rd Design Automation Conference (DAC'96)*, pages 655–660. IEEE Computer Society Press, June 1996.
3. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with blast. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18, 2005.
4. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *proceedings of the 36rd Design Automation Conference (DAC'99)*. IEEE Computer Society Press, June 1999.

5. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, number 1579 in Lecture Notes in Computer Science. Springer-Verlag, 1999.
6. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theor.Comp.Science*, 59(1–2), July 1988.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 35(8):677–691, 1986.
8. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
9. Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.
10. P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D.Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD)*, November 2002.
11. Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.
12. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.
13. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.
14. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
15. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1992.
16. Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.
17. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *J. ACM*, 50(5):752–794, 2003.
18. E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine leraning techniques. In *Proc. of Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–150, Copenhagen, Denmark, July 2002. Springer-Verlag.
19. E.M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS)*, Copenhagen, Denmark, July 2002.
20. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, , and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, Paris, France, July 2001.
21. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

22. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and System (TOPLAS)*, 19(2), 1997.
23. Dennis Dams and Kedar S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Logic in Computer Science (LICS)*, pages 335–344, 2004.
24. Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39, Boulder, CO, USA, July 2003.
25. E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *J. ACM*, 33(1):151–178, 1986.
26. R. Fraer, G. Kamhi, Z. Barukh, M.Y. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In *12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, Chicago, USA, July 2000.
27. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *DATE*, 2002.
28. Sussanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. of Conference on Computer-Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.
29. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
30. O. Grumberg, A. Schuster, and A. Yadgar. Reachability using a memory-efficient all-solutions sat solver. In *Fifth Inernation Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, November 2004.
31. Z. Har'El and R. P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, Jan.–Feb. 1990.
32. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
33. G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editors, 1991.
34. G. Holzmann. Logic verification of ansi-c code with SPIN. In *Proceedings of the 7th international SPIN workshop*, volume 1885 of *LNCS*, pages 131–147, 2000.
35. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.
36. R. P. Kurshan. *Computer-Aided Verification of coordinating processes - the automata theoretic approach.* Princeton University Press, Princeton, New Jersey, 1994.
37. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–45, 1995.
38. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning.* PhD thesis, Carnegie Mellon University, 1993.
39. J.P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
40. K. McMillan. Applications of craig interpolation to model checking. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 1–12, Edinburgh, Scotland, April 2005. Springer.

41. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.
42. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
43. K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, 2003.
44. Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification*, 2002.
45. R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489. BCS, 1971.
46. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second Internation Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.
47. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *39th Design Aotomation Conference (DAC'01)*, 2001.
48. D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.
49. A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science*, 13:45–60, 1981.
50. Hassen Saidi and Natarajan Shankar. Abstract and model check while you prove. In *Proceedings of the eleventh International Conference on Computer-Aided Verification (CAV99)*, Trento, Italy, July 1999.
51. M. Sheeran, S. Singh, and G. Staalmarck. Checking safety properties using induction and a sat-solver. In *Third International Conference on Formal methods in Computer-Aided Design (FMCAD'00)*, Austin, Texas, November 2000.
52. M. Sheeran and G. Staalmarck. A tutorial on stalmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1), January 2000.
53. Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinemnet. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 275–287, Boulder, CO, USA, July 2003. Springer.