ABSTRACTIONS AND REDUCTIONS IN MODEL CHECKING

ORNA GRUMBERG
Computer Science Department
The Technion
Haifa 32000
Israel

Abstract. We introduce the basic concepts of *temporal logic model checking* and its *state explosion problem*. We then focus on *abstraction*, which is one of the major methods for overcoming this problem. We distinguish between weak and strong preservations of properties by a given abstraction. We show how abstract models preserving ACTL* can be defined with human aid or automatically. When the abstraction is too coarse, we show how refinement can be applied to produce a more precise abstract model. Abstract interpretation is then introduced and applied in order to construct abstract models that are more precise and allow more ACTL properties to be proven. Finally, we show how to define abstract models that preserve ECTL* and full CTL*.

Keywords: model checking, temporal logic, abstraction, refinement, abstract interpretation, bisimulation, simulation preorder

1. Introduction

Temporal logic model checking is a procedure that gets as input a finite state model for a system and a property written in propositional temporal logic. It returns "yes" if the system has the desired property and returns "no" otherwise. In the latter case it also provides a counterexample that demonstrates how the system fails to satisfy the property. Model checking procedures can be quite efficient in time. However, they suffer from the state explosion problem: the number of states in the model grows exponentially with the number of system variables and components. This problem

impedes the applicability of model checking to large systems, and much effort is invested in trying to avoid it.

Abstraction is a method for reducing the state space of the checked system. The reduction is achieved by hiding (abstracting away) some of the system details that might be irrelevant for the checked property. Abstract models are sometimes required to *strongly preserve* the checked properties. In this case, a property holds on the abstract model if and only if it holds on the original one. On the other hand, only *weak preservation*, may be required. In that case, if a property is true for the abstract model then it is also true for the original model. If a property is not true for the abstract model, then no conclusion can be reached with regards to the original model. The advantage of weak preservation is that it enables more significant reductions. However, it also increases the likelihood that we will be unable to determine the truth of a property in the system.

The decision as to which details are unnecessary for the verification task is made either manually (by the verification engineer) or automatically. In both cases, if the abstract model cannot determine the truth of the property in the system, then *refinement* is applied and additional details are introduced into the model.

The main goal of abstraction is to avoid the construction of the full system model. Thus, we need methods that derive an abstract model directly from some high-level description of the system (e.g. program text).

We will define conditions for strong and weak preservations for the temporal logic CTL* and its universal and existential fragments, ACTL* and ECTL*. We will show how to derive abstract models which preserve ACTL* from the program text, using nonautomatic and automatic abstractions. When the abstractions are too coarse, we will show how they can be refined.

The basic notions of abstract interpretation will be defined, and its use for deriving abstract models will be demonstrated. Abstract interpretation provides means for constructing more precise abstract models that allow us to prove more ACTL* properties. Within the framework of abstract interpretation we also show how to define abstract models that preserve ECTL* and full CTL*.

The rest of the paper is organized as follows. Section 2 defines temporal logics along with their semantics. It presents a model checking algorithm for CTL, and defines the notions of equivalence and preorder on models. Section 3 describes data abstraction. Approximated abstractions are also defined. It then shows how abstractions and approximations can be derived from a high level description of the program. Section 4 presents the ideas of counterexample-guided refinement in which both the initial abstraction and the refinement are constructed automatically. Section 5 develops abstract

models within the abstract interpretation framework. Finally, Section 6 reviews the related work and Section 7 presents some concluding remarks.

2. Preliminaries

2.1. TEMPORAL LOGICS

We use finite state transition systems called *Kripke models* in order to model the verified systems.

Definition 2.1 (Kripke model) *Let AP be a set of atomic propositions. A* Kripke model *M over AP is a four-tuple $M = (S, S_0, R, L)$, where*

- *S is the 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.

We use propositional temporal logics as our specification languages. We present several subsets of the temporal logic CTL* [30] over a given finite set AP of atomic propositions. We will assume that formulas are expressed in *positive normal form*, in which negations are applied only to atomic propositions. This facilitates the definition of universal and existential subsets of CTL* [36]. 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 $\mathbf{R}$ (sometimes called the *release* operator), which is the dual of the *until* operator $\mathbf{U}$, is also added. Thus, $\neg(f\ \mathbf{U}\ g) = \neg f\ \mathbf{R}\ \neg g$.

Definition 2.2 (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 $f\ \mathbf{R}\ g$.*

The abbreviations $true, false$ *and implication* $\rightarrow$ *are defined as usual. For path formula* f*, we also use the abbreviations* $\mathbf{F}\, f \equiv true\, \mathbf{U}\, f$ *and* $\mathbf{G}\, f \equiv false\, \mathbf{R}\, f$*. They express the properties that* sometimes in the future f *will hold and 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, and no nesting of temporal operators is allowed. More precisely, CTL 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 $\mathbf{AX}\, f$, $\mathbf{A}(f \mathbf{U} g)$, $\mathbf{A}(f \mathbf{R} g)$ and $\mathbf{EX}\, f$, $\mathbf{E}(f \mathbf{U} g)$, $\mathbf{E}(f \mathbf{R} g)$.

ACTL* and ECTL* (*universal* and *existential* CTL*) are subsets of CTL* in which the only allowed path quantifiers are $\mathbf{A}$ and $\mathbf{E}$, respectively. ACTL and ECTL are the restriction of ACTL* and ECTL* to CTL.

LTL [55] 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 and g are path formulas, then $f \wedge g$, $f \vee g$, $\mathbf{X}\, f$, $f\, \mathbf{U}\, g$, and $f\, \mathbf{R}\, g$ are path formulas.

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

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

Definition 2.3 (Satisfaction of a formula) *Given a Kripke model* 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 the state formula.*
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) $\pi \models f\,\mathbf{R}\,g$ *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 [30] it has been shown that CTL and LTL are incomparable in their expressive power, and that CTL* is more expressive than each.

Below we present several formulas together with their intended meaning and their syntactic association with some of the logics mentioned above.

- **Mutual exclusion:** $\mathbf{AG}\neg(cs_1 \wedge cs_2)$ where cs_i is an atomic proposition that is true in a state if and only if process i is in its critical section in that state. The formula means that it is invariantly true (in every state along every path) that processes 1 and 2 cannot be in their critical section at the same time.
  The formula is in CTL, LTL and CTL*.
- **Nonstarvation:** $\mathbf{AG}(request \longrightarrow \mathbf{AF}\,grant)$ means that every request will be granted along every execution.
  This formula is in CTL but not in LTL. However, it is equivalent to the LTL formula $\mathbf{AG}(request \longrightarrow \mathbf{F}\,grant)$.
- **"Sanity check":** The formula $\mathbf{EF}\,request$ is complementary to the nonstarvation formula. It excludes the case where the implication in the nonstarvation formula holds vacuously just because no request has been presented.
  This is a CTL formula that is not expressible in LTL.
- **Fairness:** $\mathbf{A}(\mathbf{GF}\,enabled \longrightarrow \mathbf{GF}\,executed)$ describes a fairness requirement that a transition which is infinitely often enabled ($\mathbf{GF}\,enabled$) is also infinitely often executed ($\mathbf{GF}\,executed$).
  This LTL formula is not expressible in CTL.
- **Reaching a reset state:** $\mathbf{AGEF}\,reset$ describes a situation where in every state along every path there is a possible continuation that will eventually reach a reset state.
  The formula is in CTL and is not expressible in LTL.

2.2. CTL MODEL CHECKING

In this section we briefly describe an algorithm for CTL model checking. CTL model checking [29] is widely used due to its efficient algorithm. LTL model checking [44] is also commonly used because many useful properties are easily expressed in LTL. CTL* model checking [31] can be built as a combination of the algorithms for LTL and CTL. For more details on these algorithms see [17].

The CTL model checking algorithm receives a Kripke model $M = (S, S_0, R, L)$ and a CTL formula f. It works iteratively on subformulas of f, from simpler subformulas to more complex ones. For each subformula

g of f, it returns the set S_g of all states in M that satisfy g. That is, $S_g = \{s|\ M, s \models g\ \}$. An important property of the algorithm is that when it checks the formula g, all subformulas of g have already been checked. When the algorithm terminates, it returns $True$ if $S_0 \subseteq S_f$ and returns $False$ otherwise.

For CTL model checking, positive normal form is not necessary, in which case every formula can be expressed using the Boolean operators $\neg$ and $\wedge$ and the temporal operators **EX**, **EU**, and **EG**. The model checking algorithm consists of several procedures, each taking care of formulas in which the main operator is one of the above. Here, we present only the more complex procedures for formulas of the form $f = \mathbf{E}(g\mathbf{U}h)$ and $f = \mathbf{EG}\, g$.

The procedure CheckEU, presented in Figure 1, accepts as input the sets of states S_g and S_h and iteratively computes the set S_f of states that satisfy $f = \mathbf{E}(g\mathbf{U}h)$. The computation is based on the equivalence

$$\mathbf{E}(g\mathbf{U}h) = h\ \vee (g\ \wedge\ \ \mathbf{EX}(\mathbf{E}(g\mathbf{U}h)).$$

At each iteration, Q holds the set of states computed in the previous iteration while Q' holds the set of states computed in the current iteration. Initially, all states that satisfy h are introduced into Q'. At step i, all states in S_g that have a successor in Q are added. These are exactly the states that satisfy g and have a successor that satisfies $\mathbf{E}(g\mathbf{U}h)$. The computation stops when no more states can be added, i.e., a fixpoint is reached (in fact, this is a *least fixpoint*).

procedure CheckEU(S_g, S_h)
$Q := \emptyset$; $Q' := S_h$;
while $(Q \neq Q')$ **do**
    $Q := Q'$;
    $Q' := Q \cup \{s \mid \exists s'[\ R(s,s') \wedge Q(s') \wedge S_g(s)\ ]\ \}$;
end while;
$S_f := Q$; **return** (S_f)

Figure 1. The procedure CheckEU for checking the formula $f = \mathbf{E}(g\mathbf{U}h)$

The procedure CheckEG, presented in Figure 2, accepts the set of states S_g and iteratively computes the set S_f of states that satisfy $f = \mathbf{EG}\, g$. The computation is based on the equivalence

$$\mathbf{EG}\, g = g\ \wedge\ \ \mathbf{EX}(\mathbf{EG}\, g).$$

Initially, all states that satisfy g are introduced into Q'. At any step, states that do not have a successor in Q are removed. When the computation terminates, each state in Q has a successor in Q. Since all states in Q satisfy g, they all satisfy $\mathbf{EG}\, g$. In this case, a *greatest fixpoint* is computed.

procedure CheckEG(S_g)
$Q := S$; $Q' := S_g$;
while $(Q \neq Q')$ **do**
    $Q := Q'$;
    $Q' := Q \cap \{s \mid \exists s'[\, R(s,s') \wedge Q(s') \,]\ \}$;
end while;
$S_f := Q$; **return** (S_f)

Figure 2. The procedure CheckEG for checking the formula $f = \mathbf{EG}\, g$

Theorem 2.4 *[17] Given a model M and a CTL formula f, there is a model checking algorithm that works in time $O((|S| + |R|) \cdot |f|)$.*

2.3. EQUIVALENCES AND PREORDERS

In this section we define the bisimulation relation and the simulation preorder over Kripke models. We will also state the relationships between these relations and logic preservation. 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 models over AP.

Definition 2.5 *A relation $B \subseteq S_1 \times S_2$ is a* bisimulation relation *[54] 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 $s_1 \equiv s_2$ for $B(s_1, s_2)$. We say that M_1 and M_2 are *bisimilar* (denoted $M_1 \equiv M_2$) if there exists a bisimulation relation B over M_1 and M_2.

Definition 2.6 *A relation $H \subseteq S_1 \times S_2$ is a* simulation relation *[51] 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 relation $\equiv$ is an equivalence relation on the set of models, while the relation $\preceq$ is a preorder on this set. That is, $\equiv$ is reflexive, symmetric and transitive and $\preceq$ is reflexive and transitive. Note that if there is a bisimulation relation over M_1 and M_2, then there is a *unique* maximal bisimulation relation over M_1 and M_2, that includes any other bisimulation relation over M_1 and M_2. A similar property holds also for simulation.

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

Theorem 2.7

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

The last part of Theorem 2.7 is a direct consequence of the previous part of this theorem. It is based on the observation that for every ECTL* formula f, there is an ACTL* formula which is equivalent to $\neg f$.

3. Data Abstraction

The first abstraction that we present is *data abstraction* [15, 47]. In order to obtain a smaller model for the verified system, we abstract away some of the data information. At the same time, we make sure that each behavior of the system is represented in the reduced model. In fact, the reduced model may contain more behaviors than the concrete (full) model. Nevertheless, it is often smaller in size (number of states and transitions), which makes it easier to apply model checking.

Data abstraction is done by choosing, for every variable in the system, an abstract domain that is typically significantly smaller than the original domain. The abstract domain is chosen by the user. The user also supplies

[1]Bisimulation and simulation also preserve the μ-calculus logic [40] and its universal [46] and existential subsets, respectively. The discussion of this is beyond the scope of this paper.

a mapping from the original domain onto the abstract one. An abstract model can then be defined in such a way that it is *greater by the simulation preorder* than the concrete model of the system. Theorem 2.7 can now be used to verify ACTL* properties of the system by checking them on the abstract model.

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 $\widehat{x_i} = a$. These atomic formulas indicate that the variable x_i has some value d that has been abstracted to a.

The definition of the abstract model is based on the definition of the concrete model. However, building it on top of the concrete model would defeat the purpose since the concrete model is often too large to fit into memory. Instead, we show how the abstract model can be derived directly from some high-level description of the program, e.g., from the program text. As we will see later, extracting a precise abstract model may not be an easy task. We therefore define an *approximated abstract model*. This model may have more behaviors than the abstract model, but it is easier to build from the program text.

Let P be a program with variables $x_1, \ldots, x_n$. For simplicity we assume that all variables are over the same domain D. Thus, the concrete (full) 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 x_i in this state (denoted $s(x_i) = d_i$).

The first step in building an abstract model for P is choosing an abstract domain A and a surjection $h : D \to A$. The next step is to restrict the concrete model of P so that it reflects only the abstract values of its variables. This is done by defining a new set of atomic propositions

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

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

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

Example 3.1 *Let 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:

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

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

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 model induced by* A_1 *and* h_1 *is:* $L_1(s) = \{\widehat{x} = a_+\}$ *and* $L_1(s') = \{\widehat{x} = a_-\}$.

Abstraction 2:

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

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

The set of atomic propositions is $AP_2 = \{\ \widehat{x} = a_{even},\ \widehat{x} = a_{odd}\ \}$. *The labeling of states in the concrete model induced by* A_2 *and* h_2 *is:* $L_2(s) = \{\widehat{x} = a_{even}\}$ *and* $L_2(s') = \{\widehat{x} = a_{odd}\}$.

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.

Given A and h as above, we can now define the most precise abstract model, M_r, called the *reduced model*. First we extend the mapping $h : D \to A$ to n-tuples in $D \times \ldots \times D$:

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

An abstract state $(a_1, \ldots, a_n)$ of M_r will represent the set of all states $(d_1, \ldots, d_n)$ such that $h((d_1, \ldots, d_n)) = (a_1, \ldots, a_n)$. Concrete states s_1, s_2 are said to be *equivalent* ($s_1 \sim s_2$) if and only if $h(s_1) = h(s_2)$, that is, both states are mapped to the same abstract state. Thus, each abstract state represents an equivalence class of concrete states.

Definition 3.1 *Given a concrete model* M, *an abstract domain* A, *and an abstraction mapping* $h : D \to A$, *the* reduced model $M_r = (S_r, S_{0_r}, R_r, L_r)$ *is defined as follows:*

- $S_r = A \times \ldots \times A$.
- $S_{0_r}(s_r) \Leftrightarrow \exists s \in S_0 : h(s) = s_r$.
- $R_r(s_r, t_r) \Leftrightarrow \exists s, t[\ h(s) = s_r \wedge h(t) = t_r \wedge R(s,t)\ ]$.
- *For* $s_r = (a_1, \ldots, a_n)$, $L_r(s_r) = \{\ \widehat{x_i} = a_i \mid i = 1, \ldots, n\ \}$.

This type of abstraction is called *existential abstraction*.

Lemma 3.2 *The reduced model* M_r *is greater by the simulation preorder than the concrete model* M; *that is,* $M \preceq M_r$.

To see why the lemma is true, note that the relation $H = \{ (s, s_r) \mid h(s) = s_r \}$ is a simulation preorder between M and M_r. The following corollary is a direct consequence of this lemma and Theorem 2.7.

Corollary 3.3 *For every ACTL* formula φ, if $M_r \models \varphi$ then $M \models \varphi$.*

3.1. DERIVING MODELS FROM THE PROGRAM TEXT

In the next section we explain how the reduced and approximated model for the system can be derived directly from a high-level description of the system. 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 P be a program, and let $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{x}' = (x'_1, \ldots, x'_n)$ be two copies of the program variables, representing the current and next state, respectively. The program will be given by two first-order formulas, $\mathcal{S}_0(\overline{x})$ and $\mathcal{R}(\overline{x}, \overline{x}')$, describing the set of initial states and the set of transitions. Let $\overline{d} = (d_1, \ldots, d_n)$ be a vector of values. The notation $\mathcal{S}_0[\overline{x} \leftarrow \overline{d}]$ indicates that for every $i = 1, \ldots, n$, the value d_i is substituted for the variable x_i in the formula $\mathcal{S}_0$. A similar notation is used for substitution in the formula $\mathcal{R}$.

Definition 3.4 *Let $S = D \times \ldots \times D$ be the set of states in a model M. The formulas $\mathcal{S}_0(\overline{x})$ and $\mathcal{R}(\overline{x}, \overline{x}')$ 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.*

- *$S_0(s) \Leftrightarrow \mathcal{S}_0(\overline{x})[\overline{x} \leftarrow \overline{d}]$ is true.*
- *$R(s, s') \Leftrightarrow \mathcal{R}(\overline{x}, \overline{x}')[\overline{x} \leftarrow \overline{d}, \overline{x'} \leftarrow \overline{d'}]$ 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 [17]. 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 3.2 *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:*

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

The statement $l: \ x := e \ \ l'$ is described by the formula:

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

The statement $l:$ *if* $x = 0$ *then* $l_1:\ x := 1$ *else* $l_2:\ x := x + 1\ l'$ *is described by the formula:*

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

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.

3.2. DERIVING ABSTRACT MODELS

Given $\mathcal{S}_0$ and $\mathcal{R}$ that describe a concrete model M, we would like to define formulas $\widehat{\mathcal{S}}_0$ and $\widehat{\mathcal{R}}$ that describe the reduced model M_r. The new formulas will be defined over variables $\widehat{x_i}$, which range over the abstract domain. 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 $\widehat{x_1}, \ldots, \widehat{x_k}$ from a formula over $x_1, \ldots, x_k$.

Definition 3.5 *Let* ϕ *be a first-order formula over variables* $x_1, \ldots, x_k$. *The formula* $[\phi]$ *over* $\widehat{x_1}, \ldots, \widehat{x_k}$ *is defined as follows:*

$$[\phi](\widehat{x_1}, \ldots, \widehat{x_k}) = \exists x_1 \ldots x_k \left(\bigwedge_{i=1}^{k} h(x_i) = \widehat{x_i} \wedge \phi(x_1, \ldots, x_k) \right).$$

Lemma 3.6 *Let* $\mathcal{S}_0$ *and* $\mathcal{R}$ *be the formulas describing a model* M *over states in* $D \times \ldots \times D$. *Then the formulas* $\widehat{\mathcal{S}}_0 = [\mathcal{S}_0]$ *and* $\widehat{\mathcal{R}} = [\mathcal{R}]$ *describe the reduced model* M_r *over* $A \times \ldots \times A$.

The lemma holds since M_r is defined by existential abstraction (see Definition 3.1). 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 reduced model M_r 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 3.7 *Let* ϕ *be a first-order formula in positive normal form. Then the following holds:*

1. *If p is a primitive relation, then* $\mathcal{T}(p(x_1,\ldots,x_k)) = [p](\widehat{x}_1,\ldots,\widehat{x}_k)$ *and* $\mathcal{T}(\neg p(x_1,\ldots,x_k)) = [\neg p](\widehat{x}_1,\ldots,\widehat{x}_k)$.
2. $\mathcal{T}(\phi_1 \wedge \phi_2) = \mathcal{T}(\phi_1) \wedge \mathcal{T}(\phi_2)$.
3. $\mathcal{T}(\phi_1 \vee \phi_2) = \mathcal{T}(\phi_1) \vee \mathcal{T}(\phi_2)$.
4. $\mathcal{T}(\forall x \phi) = \forall \widehat{x} \mathcal{T}(\phi)$.
5. $\mathcal{T}(\exists x \phi) = \exists \widehat{x} \mathcal{T}(\phi)$.

We can now define an *approximated* abstract model M_a. It is defined over the same set of states as the reduced 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 M_r is also an initial state of M_a. Moreover, every transition of M_r is also a transition of M_a.

Lemma 3.8 *For every first-order formula ϕ in positive normal form, $[\phi]$ implies $\mathcal{T}(\phi)$. 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 3.7 result in nonequivalent formulas.

Corollary 3.9 $M \preceq M_r \preceq M_a$.

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

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

In [15, 47] several data abstractions have been suggested and used to verify meaningful properties of interesting programs.

4. Counterexample-Guided Abstraction Refinement

In the previous section we showed how an abstract model can be constructed based on an abstract domain and a mapping, both provided by the user. Unfortunately, choosing a suitable abstraction is not trivial for large systems and requires considerable creativity.

In this section we describe a technique (presented in [16]) that determines the required domain and mapping automatically, based on an analysis of the program text. This technique differs from data abstraction in that the abstract domains and abstract mappings are defined for *clusters of variables* rather than single variables. The clusters are chosen according

to dependencies that are found among the variables in the program. Abstracting clusters of variables results in an abstract model that reflects the system behavior more precisely.

As in the previous section, we use existential abstraction. Existential abstraction guarantees that ACTL* properties true of the abstract model are also true of the concrete model. However, if a property is false in the abstract model, then the counterexample produced by the model checking algorithm may be the result of some behavior that is not present in the concrete model. In this case, we refine the abstraction to eliminate the erroneous behavior from the abstract model. The refinement is determined by information obtained from the counterexample.

The suggested method has the following steps:

- **Generating an initial abstraction**.
  This involves the construction of variable clusters for variables which interfere with each other via conditions in the program.
- **Model-checking the abstract model**.
  If the formula is true, we conclude that the concrete model satisfies the formula and stop. If a counterexample $\widehat{T}$ is found, we check whether $\widehat{T}$ is a counterexample in the concrete model. If it is, we conclude that the concrete model does not satisfy the formula and stop. Otherwise, $\widehat{T}$ is a spurious counterexample, and we proceed to step 3.
- **Refining the abstraction**.
  This is done by splitting one abstract state so that $\widehat{T}$ is not included in the new abstract model. We then go back to step 2.

4.1. ASSUMPTIONS

There are several assumptions that are needed in order to make our method fully automatic and effective.

- The program is finite-state, i.e., each variable is over a finite domain.
- We use BDD-based algorithms. BDD [10] is a data structure for representing Boolean functions. BDDs are often very concise in their space requirements. Sets of states and sets of transitions of Kripke models can easily be represented by BDDs. Moreover, most operations applied in model checking algorithms can be implemented efficiently with BDDS. As a result, BDD-based model checking [13, 49] (called *symbolic model checking*) is very useful in practice.
- The full model is too large to fit into memory, even when represented by BDDs. However, subsets of the full state space can be held in memory. In particular, we will maintain as BDDs the sets of concrete states that are mapped to a specific abstract state.

– The transition relation of the full model is available. If it is too large, it is held *partitioned*. There are known techniques for handling partitioned transition relations in model checking [12].

4.2. GENERATING THE INITIAL ABSTRACTION

Let P be a program over variables $x_1, \ldots, x_n$. Suppose that each variable x_i is defined over domain D_{x_i}, which is finite. Finiteness of the domains is necessary in order for the method to be fully automatic. Let φ be the ACTL* formula to be checked on P.

Atomic formulas will be defined over program variables, constants and relation symbols. For instance, $x > y$ and $x = 1$ are atomic formulas. Boolean combinations of atomic formulas are used as conditions in the program. The logic ACTL* is also defined over atomic formulas of this type. We use *Atom* to denote the set of all atomic formulas appearing in P and φ.

Given a state $s = \overline{d} = (d_1, \ldots, d_n)$ and an atomic formula p over $x_1, \ldots, x_n$, we write $s \models p$ if $p[\overline{x} \leftarrow \overline{d}] = true$. That is, p is evaluated to $true$ when each variable x_i is assigned the value d_i.

As before, the concrete model M of program P is defined over states $S = D_{x_1} \times \ldots \times D_{x_n}$, where each state in M is labeled with the set of atomic formulas from *Atom* that are true in that state.

We say that two atomic formulas *interfere* if the sets of variables appearing in them are not disjoint. Let $\equiv_I$ be that equivalence relation over *Atom* which is the reflexive, transitive closure of the interference relation. The equivalence class of an atomic formula $f \in Atom$ is called the *cluster of* f and is denoted by $[f]$. Note that if two atomic formulas f_1 and f_2 have nondisjoint sets of variables, then $[f_1] = [f_2]$. That is, a variable cannot occur in formulas that belong to different formula clusters.

Consequently, we can define an equivalence relation $\equiv_V$ on the program variables as follows:

$$x_i \equiv_V x_j \text{ if and only if } x_i \text{ and } x_j \text{ appear in atomic formulas that belong to the same formula cluster.}$$

The equivalence classes of $\equiv_V$ are called variable clusters. Let $\{FC_1, \ldots, FC_m\}$ and $\{VC_1, \ldots, VC_m\}$ be the set of formula clusters and variable clusters, respectively. Each variable cluster VC_i is associated with a domain $D_{VC_i} = \prod_{x \in VC_i} D_x$, representing the value of all the variables in this cluster. Note that $S = D_{VC_1} \times \ldots \times D_{VC_m}$.

Example 4.1 *Let* $Atom = \{\ x > y,\ x = 1,\ z = t\}$. *Then there are two formula clusters,* $FC_1 = \{\ x > y,\ x = 1\ \}$ *and* $FC_2 = \{\ z = t\ \}$, *and two variable clusters,* $VC_1 = \{x, y\}$ *and* $VC_2 = \{z, t\}$.

The initial abstraction is defined by $h = (h_1, \ldots, h_m)$, where h_i is defined over D_{VC_i} as follows. For $(d_1, \ldots, d_k), (e_1, \ldots, e_k) \in D_{VC_i}$,

$$h_i((d_1, \ldots, d_k)) = h_i((e_1, \ldots, e_k)) \iff$$

$$\forall f \in FC_i : (d_1, \ldots, d_k) \models f \Leftrightarrow (e_1, \ldots, e_k) \models f.$$

Thus, two states are h_i-equivalent if and only if they satisfy the same formulas in FC_i. They are h-equivalent if and only if they satisfy the same formulas in *Atom*.

The reduced model for the abstraction h will be defined over abstract states that are the equivalence classes of the h-equivalence. Each equivalence class will be labeled by all formulas from *Atom* which are true in all states in the class. The initial states and transition relation are defined by the existential abstraction, as before.

Example 4.2 *Let P be a program with variables x, y over domain $\{0,1\}$ and z, t over domain $\{true, false\}$. Let $FC_1 = \{\ x > y,\ x = 1\ \}$, $FC_2 = \{\ z = t\ \}$, $VC_1 = \{x, y\}$ and $VC_2 = \{z, t\}$. Then, the h_1-equivalence classes are:*

$$E_{11} = \{(0,0),(0,1)\},\ E_{12} = \{(1,0)\},\ E_{13} = \{(1,1)\}.$$

The h_2-equivalence classes are:

$$E_{21} = \{(false, false), (true, true)\},\ E_{22} = \{(false, true), (true, false)\}.$$

The reduced model contains six states labeled by:

$$\begin{aligned}
L_r((E_{11}, E_{21})) &= \{z = t\} \\
L_r((E_{12}, E_{21})) &= \{x > y, x = 1, z = t\} \\
L_r((E_{13}, E_{21})) &= \{x = 1, z = t\} \\
L_r((E_{11}, E_{22})) &= \emptyset \\
L_r((E_{12}, E_{22})) &= \{x > y, x = 1\} \\
L_r((E_{13}, E_{22})) &= \{x = 1\}
\end{aligned}$$

Example 4.2 shows that abstracting variable clusters rather than single variables allows smaller abstract domains that are more precise. Valuations of a whole variable cluster determine the truth of the conditions in the program, and thus influence its control flow. Abstracting a whole cluster allows us to identify all states that satisfy the same conditions and then abstract them together. Thus, the abstract model reflects more closely the control flow of the program.

4.3. IDENTIFYING SPURIOUS PATH COUNTEREXAMPLES

Once the reduced model is built, we can run the model checking algorithm on it. Suppose the model checking stopped with a path counterexample. Such a counterexample is produced, for instance, when the checked property is $\mathbf{AG}\,p$ for some atomic formula p. The path leads from an initial state to a state that falsifies p. Our goal is to find whether there is a corresponding path in the concrete model from an initial state to a state that falsifies p.

Let $\widehat{T} = \widehat{s_1} \ldots \widehat{s_n}$ be the path counterexample in the reduced model. For an abstract state $\widehat{s}$, $h^{-1}(\widehat{s})$ denotes the set of concrete states that are mapped to $\widehat{s}$, i.e., $h^{-1}(\widehat{s}) = \{\ s \mid h(s) = \widehat{s}\ \}$. $h^{-1}(\widehat{T})$ denotes the set of all concrete paths from an initial state that correspond to $\widehat{T}$, i.e.,

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

We say that a path $\widehat{T} = \widehat{s_1} \ldots \widehat{s_n}$ *corresponds to a real counterexample* if there is a path $\pi = s_1 \ldots s_n$ such that for all $1 \leq i \leq n$, $h(s_i) = \widehat{s_i}$. Moreover, π starts at an initial state and its final state falsifies p. Note that if $h(s_i) = \widehat{s_i}$, then s_i and $\widehat{s_i}$ satisfy the same atomic formulas. Thus, it immediately follows that $\widehat{T}$ corresponds to a real counterexample if and only if $h^{-1}(\widehat{T})$ is not empty.

The algorithm SplitPATH in Figure 3 checks whether $h^{-1}(\widehat{T})$ is empty. In fact, it computes a sequence $S_1, \ldots, S_n$ of sets of states. For each i, S_i contains states that correspond to $\widehat{s_i}$ and are also successors of states in S_{i-1}. It starts with $S_1 = h^{-1}(\widehat{s_1}) \cap S_0$, which includes all initial states that correspond to $\widehat{s_1}$. If some S_i turns out to be empty, then SplitPATH returns both the place i where the failure occurred and the last nonempty set S_{i-1}.

SplitPATH uses the transition relation R of the concrete model M in order to compute $Img(S_{i-1}, R)$, which is the set of all successors of states in S_{i-1}. All operations in SplitPATH, including Img, are effectively implemented with BDDs (symbolic implementation). The following lemma proves the correctness of SplitPATH.

Lemma 4.1 *$\widehat{T}$ corresponds to a real counterexample if and only if for all $1 \leq i \leq n$, $S_i \neq \emptyset$.*

4.4. IDENTIFYING SPURIOUS LOOP COUNTEREXAMPLES

Suppose that the model checking returns a loop counterexample. This may occur, for instance, when the property $\mathbf{AF}\,p$ is checked. The counterexample exhibits an infinite path along which p never holds. A loop counterexample

Algorithm SplitPATH$(\widehat{T})$

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

Figure 3. SplitPATH checks if an abstract path is spurious.

will be of the form

$$\widehat{T} = \widehat{s_1} \ldots \widehat{s_i}\ \langle \widehat{s_{i+1}} \ldots \widehat{s_n} \rangle^{\omega},$$

where the sequence $\widehat{s_{i+1}} \ldots \widehat{s_n}$ repeats forever.

Some difficulties arise when trying to determine whether a loop counterexample in the abstract model corresponds to a real counterexample in the concrete model. First, an abstract loop may correspond to different loops of different sizes in the concrete model. Furthermore, the loops may start at different stages of the unwinding. Clearly, the unwinding must eventually result in a periodic path. However, a careful analysis is needed in order to see that a polynomial number of unwindings is sufficient. More precisely, let

$$min = \min\{\ |h^{-1}(\widehat{s_{i+1}})|, \ldots, |h^{-1}(\widehat{s_n})|\ \}.$$

That is, min is the size of the smallest set of concrete states that corresponds to one of the abstract states on the loop. Then $min + 1$ unwindings are sufficient. This is formalized in the following lemma. Let $\widehat{T} = \widehat{s_1} \ldots \widehat{s_i}\ \langle \widehat{s_{i+1}} \ldots \widehat{s_n} \rangle^{\omega}$ and $\widehat{T}_{unwind} = \widehat{s_1} \ldots \widehat{s_i}\ \langle \widehat{s_{i+1}} \ldots \widehat{s_n} \rangle^{min+1}$.

Lemma 4.2 *$\widehat{T}$ corresponds to a concrete counterexample if and only if $h^{-1}(\widehat{T}_{unwind})$ is not empty.*

Following is an intuitive explanation for the correctness of the "if" clause of the lemma. Assume $h^{-1}(\widehat{T}_{unwind}) \neq \emptyset$. For any concrete path π in $h^{-1}(\widehat{T}_{unwind})$, the suffix π^{i+1} of π goes $min + 1$ times through each of the sets $h^{-1}(\widehat{s_j})$, for $j = i+1, \ldots, n$. Suppose $min = |h^{-1}(\widehat{s_M})|$. Then at least one state in $h^{-1}(\widehat{s_M})$ repeats twice along π^{i+1}, thus forming a concrete loop. Replacing π^{i+1} with this loop in π results in a loop counterexample in the concrete model.

From this lemma we conclude that the loop counterexample can be reduced to a path counterexample. Figure 4 presents the algorithm SplitLOOP, which applies SplitPATH to $\widehat{T}_{unwind}$ in order to check if an abstract loop is spurious. **LoopIndex**(j) computes the index in the unwound $\widehat{T}_{unwind}$ of the abstract state at position j. That is,

$$\textbf{LoopIndex}(j) = \text{ if } j \leq n \text{ then } j \text{ else}((j-(i+1)) mod (n-i)) + (i+1).$$

Thus, SplitLOOP returns two indices k and p, which are consecutive on the loop, and the set $S_{prev} \subseteq h^{-1}(\widehat{s_k})$. In the refinement step, the loop counterexample can now be treated similarly to the path counterexample.

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_{prev} as in **SplitPATH**$(\widehat{T}_{\text{unwind}})$;
$k :=$ **LoopIndex**(j);
$p :=$ **LoopIndex**$(j+1)$;
output S_{prev}, k, p

Figure 4. SplitLOOP checks if an abstract loop is spurious

4.5. REFINING THE ABSTRACTION

Once we have realized that a path or a loop counterexample is spurious, we would like to eliminate it by refining our abstraction. We will only describe the refinement process for path counterexamples. The treatment of loop counterexamples is similar. Let j, S_{prev} be the output of SplitPATH, where $S_{prev} \subseteq h^{-1}(\widehat{s_{j-1}})$. We observe that the states in $h^{-1}(\widehat{s_{j-1}})$ can be partitioned into three subsets:

- S_D: **dead-end states**, which are reachable from an initial state in $h^{-1}(\widehat{s_1})$ but have no outgoing transition to states in $h^{-1}(\widehat{s_j})$. Note that $S_{prev} = S_D$.
- S_B: **bad states**, which are not reachable but have outgoing transitions to states in $h^{-1}(\widehat{s_j})$.
- S_I: **irrelevant states**, which are neither reachable nor dead.

Since existential abstraction is used, the dead-end states induce a path to $\widehat{s_{j-1}}$ in the abstract model. The bad states induce a transition from $\widehat{s_{j-1}}$ to $\widehat{s_j}$. Thus, a spurious path leading to $\widehat{s_j}$ is obtained.

In order to eliminate this path we need to refine the abstraction mapping h so that the dead-end states and bad states do not belong to the same abstract state.

Recall that each abstract state corresponds to an h-equivalence class of concrete states. The goal is to find a refinement that keeps the number of new equivalence classes as small as possible. It turns out that when irrelevant states are present, the problem of finding the optimal refinement is NP-complete. However, if the set of irrelevant states is empty, the problem can be solved in polynomial time. A possible heuristic associates S_I with S_B and then applies refinement, which separates them from S_D. The resulting refinement is not optimal, but gives good results in practice.

We now show how the model is refined in case $S_I = \emptyset$. Recall that $S = D_{VC_1} \times \ldots \times D_{VC_m}$ and $h = (h_1, \ldots, h_m)$, where h_i is defined over D_{VC_i}. The equivalence relations $\equiv$ ($\equiv_i$) are the sets of pairs of h-equivalent (h_i-equivalent) elements. For $S_D \subseteq S$, $i \in \{1, \ldots, m\}$, and $a \in D_{VC_i}$, we define the projection set $proj(S_D, i, a)$ by:

$$Proj(S_D, i, a) = \{\ (d_1, \ldots, d_{i-1}, d_{i+1}, \ldots, d_m) \mid$$

$$(d_1, \ldots, d_{i-1}, a, d_{i+1}, \ldots, d_m) \in S_D\ \}.$$

The *refinement procedure* checks, for each i and for each pair (a, b) in $\equiv_i$, whether $proj(S_D, i, a) = proj(S_D, i, b)$. If not, then (a, b) is eliminated from $\equiv_i$. By eliminating (a, b) from $\equiv_i$, we partition the equivalence class $h^{-1}(\widehat{s_{j-1}})$. Since $proj(S_D, i, a) \neq proj(S_D, i, b)$, there are states

$$s_a = (d_1, \ldots, d_{i-1}, a, d_{i+1}, \ldots, d_m) \text{ and } s_b = (d_1, \ldots, d_{i-1}, b, d_{i+1}, \ldots, d_m)$$

such that $s_a \in S_D$ and $s_b \notin S_D$ (or vice versa). This implies that $s_b \in S_B$ (since S_I is empty), and therefore s_a and s_b should not be in the same equivalence class. Removing (a, b) from $\equiv_i$ also removes (s_a, s_b) from $\equiv$, as required.

The refinement procedure continues to refine the abstraction mapping by partitioning equivalence classes until a real counterexample is found or until the ACTL* formula holds. Since each equivalence class is finite and nonempty, the process always terminates.

Theorem 4.3 *Given a model M and an ACTL* formula φ whose counterexample is either path or loop, the refinement algorithm finds a model $\widehat{M}$ such that $\widehat{M} \models \varphi \ \Leftrightarrow\ M \models \varphi$.*

In [16], several practical improvements are presented. The method is also experimented on large examples.

5. Abstract Interpretation

In this section we show how abstractions preserving full CTL* and its universal and existential subsets can be defined within the framework of *abstract interpretation*. We will see that abstract interpretation can be used to obtain abstract models which are more precise and therefore preserve more properties than the existential abstraction presented previously [23, 22]. The abstraction suggested in this section abstracts the state space rather than the data domain and can be applied to infinite as well as finite sets. It actually preserves the full μ-calculus (see [22] for more details).

Given an abstract domain, abstract interpretation provides a general framework for automatically "interpreting" systems on the abstract domain. The classical abstract interpretation framework [19] was used to prove safety properties, and does not consider temporal logic or model checking. Hence, it usually abstracts sets of states. Here, on the other hand, we are interested in properties of computations. We therefore abstract Kripke models, including their sets of states and transitions.

The models we use in this section are slightly different from the Kripke models presented in Definition 2.1. Instead of using the set AP of atomic propositions to label states, we use *literals* from the set

$$Lit = \ AP \ \cup \ \{\neg p \mid p \in AP \ \}.$$

The Kripke model $M = (S, S_0, R, L)$ is defined as before for S, S_0, and R. The labeling function $L : S \to \mathcal{P}(Lit)$ is required to satisfy

$$p \in L(s) \Rightarrow \neg p \notin L(s) \text{ and } \neg p \in L(s) \Rightarrow p \notin L(s).$$

Recall that labeling a state with a formula means that the formula is true in that state. Thus, by the above, p and $\neg p$ cannot be true together in a state. It is not required, however, that $p \in L(s) \Leftrightarrow \neg p \notin L(s)$. Hence, it is possible that neither p nor $\neg p$ will be true in s.

It is straightforward to extend the semantics of CTL* formulas for these models. Only for literals should it be changed. The remaining semantics is identical to Definition 2.3. For literals, we can define the semantics by

1. If $p \in AP$ then
   $s \models p$ if and only if $p \in L(s)$; $s \models \neg p$ if and only if $\neg p \in L(s)$.

As a result of this change, however, the semantics of any CTL* formula may now be such that neither $s \models \varphi$ nor $s \models \neg\varphi$.

We will now present some basic notions required for the development of an abstract interpretation framework in the context of Kripke models and temporal logic specifications. For more details see [22, 24].

Abstract interpretation assumes two partially ordered sets, $(C, \sqsubseteq)$ and $(A, \leq)$, of the concrete and abstract domains. In addition, two mappings are

used: The abstraction mapping $\alpha : C \to A$ and the concretization mapping $\gamma : A \to C$.

Definition 5.1 *$(\alpha : C \to A, \gamma : A \to C)$ is a* Galois connection *from $(C, \sqsubseteq)$ to $(A, \leq)$ if and only if*

- *α and γ are total and monotonic.*
- *for all $c \in C$, $\gamma(\alpha(c)) \sqsupseteq c$.*
- *for all $a \in A$, $\alpha(\gamma(a)) \leq a$.*

(α, γ) is a Galois insertion *if, in addition, the order $\leq$ on A is defined by the order $\sqsupseteq$ on C as follows:*

$$a \leq a' \;\Leftrightarrow\; \gamma(a) \sqsubseteq \gamma(a').$$

Note that the requirement for Galois insertion is stronger than the requirement for monotonicity of γ. Assume that $\leq$ and $\sqsupseteq$ are partial orders in which two elements are *equal* if and only if each is smaller than the other. Then, $\gamma(a_1) = \gamma(a_2)$ implies $a_1 = a_2$. Therefore, for Galois insertion

$$\alpha(\gamma(a)) = a.$$

For $a \leq a'$ we say that a is *more precise* than a', or a' *approximates* a. Similarly, for $c \sqsubseteq c'$.

We now present our framework in which the choice of concrete and abstract domains is motivated by the goal of model abstraction. Given a model M, we choose $(\mathcal{P}(S), \subseteq)$ as the concrete domain. We choose a set of abstract states $\widehat{S}$ and use the Galois insertion so that the partial order for $\widehat{S}$ is determined by

$$a \leq a' \;\Leftrightarrow\; \gamma(a) \subseteq \gamma(a').$$

Since γ and α are total and monotonic, $\widehat{S}$ must include a greatest element *top*, denoted $\top$, so that $\alpha(S) = \top$ and $\gamma(\top) = S$. Moreover, for every $S' \subseteq S$, there must be $a \in \widehat{S}$ such that $\alpha(S') = a$. However, associating each subset of concrete states with an abstract state does not imply that there must be a *different* abstract state for each subset.

Example 5.1 *A correct (though uninteresting) abstraction chooses $\widehat{S} = \{\top\}$ with $\gamma(\top) = S$ and for all $S' \subseteq S$, $\alpha(S') = \top$.*

The following abstraction is more interesting.

Example 5.2 *Let S be the set of all states with one variable x over the natural numbers. Let $\widehat{S} = \{\, grt\_5,\ leq\_5,\ \top \,\}$ where $\gamma(grt\_5) = S_{>5} = \{\, s \in S | s(x) > 5 \,\}$ and $\gamma(leq\_5) = S_{\leq 5} = \{\, s \in S | s(x) \leq 5 \,\}$. (Also, $\gamma(\top) = S$, as is always the case).*

In order to guarantee that $\alpha(\gamma(a)) = a$, *we must define* $\alpha(S_{>5}) = grt\_5$ *and* $\alpha(S_{\leq 5}) = leq\_5$. *This also guarantees that* $\gamma(\alpha(S_{>5}) \supseteq S_{>5}$ *and* $\gamma(\alpha(S_{\leq 5}) \supseteq S_{\leq 5}$.

Consider now the set $S_{>6} = \{\ s \in S | s(x) > 6\ \}$. *Both* $\alpha(S_{>6}) = grt\_5$ *and* $\alpha(S_{>6}) = \top$ *will satisfy* $\gamma(\alpha(S_{>6}) \supseteq S_{>6}$. *However, more precise abstraction is desired. Thus, since* $grt\_5 \leq \top$, *we choose* $\alpha(S_{>6}) = grt\_5$.

On the other hand, for $S_{>2} = \{\ s \in S | s(x) > 2\ \}$ *the only correct choice is* $\alpha(S_{>2}) = \top$.

Remark: The Galois insertion is less restrictive than existential abstraction in the sense that it allows nondisjoint subsets of states to be mapped to different abstract states. Furthermore, concrete states mapped to the same abstract state do not necessarily satisfy the same atomic formulas. In contrast, existential abstraction partitions the concrete state space into disjoint equivalence classes, so that all states in the same class satisfy the same set of atomic formulas.

5.1. THE ABSTRACT MODEL

The abstraction and concretization mappings defined so far determine the set of abstract states and their relationship with the set of concrete states. In order to define the abstract model we still need to define the state labeling, the set of initial states, and the transition relation of the abstract model. We start with a definition that will be used when we present the abstract transition relation.

Definition 5.2 *Let* A *and* B *be sets and* $R \subseteq A \times B$. *The relations* $R^{\exists\exists}, R^{\forall\exists} \subseteq \mathcal{P}(A) \times \mathcal{P}(B)$ *are defined as follows:*

- $R^{\exists\exists} = \{\ (X,Y) \mid \exists x \in X\ \exists y \in Y :\ R(x,y)\ \}$
- $R^{\forall\exists} = \{\ (X,Y) \mid \forall x \in X\ \exists y \in Y :\ R(x,y)\ \}$

If R is a transition relation and X and Y are subsets of states, then $R^{\exists\exists}(X,Y)$ if and only if some state in X can make a transition to some state in Y. $R^{\forall\exists}(X,Y)$ if and only if every state in X can make a transition to some state in Y. Note that the transition relation defined by existential abstraction can be viewed as an $R^{\exists\exists}$ relation over the equivalence classes represented by the abstract states.

Given a set $\widehat{S}$ of abstract states, our goal is to define a precise abstract model $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L})$ such that for every ACTL* formula φ over atomic formulas in Lit, and for every abstract state $a \in \widehat{S}$, the following requirement holds:

$$\widehat{M}, a \models \varphi \implies M, \gamma(a) \models \varphi. \tag{1}$$

5.1.1. The Abstract Labeling Function

The abstract labeling function $\widehat{L}$ is defined so that Requirement (1) holds for the literals in Lit. For every $p \in Lit$,

$$p \in \widehat{L}(a) \iff \forall s \in \gamma(a) : \; p \in L(s).$$

Thus, an abstract state is labeled by literal p if and only if all states in its concretization are labeled by p. However, since our abstraction mapping does not require that all these states be identically labeled, it is possible that neither $p \in \widehat{L}(a)$ nor $\neg p \in \widehat{L}(a)$.

Explicitly labeling the negation of atomic formulas allows us to distinguish between the case in which "all concrete states do not satisfy p" and the one in which "not all concrete states satisfy p."

The following lemma states that less precise states satisfy fewer literals. Consequently, it is desirable to map subsets of states to their most precise abstraction (see Example 5.2).

Lemma 5.3 *For $a, a' \in \widehat{S}$, if $a' \geq a$ then $\forall p \in Lit$, $a' \models p \Rightarrow a \models p$.*

5.1.2. The Abstract Initial states

The set of initial abstract states is defined by

$$\widehat{S_0} = \{\alpha(\{s\}) \mid s \in S_0 \}.$$

This guarantees Requirement (1) on the level of models. That is, $\widehat{M} \models \varphi \;\Rightarrow\; M \models \varphi$. To see why this is true, note that

$$\widehat{M} \models \varphi \implies \forall a \in \widehat{S_0} : \widehat{M}, a \models \varphi \implies$$

$$\forall a \in \widehat{S_0} : M, \gamma(a) \models \varphi \implies \forall s \in S_0 : M, s \models \varphi \implies M \models \varphi.$$

As alternative definition might be $\widehat{S_0} = \alpha(S_0)$. It also satisfies Requirement (1). However, for each $s \in S_0$, $\alpha(\{s\}) \leq \alpha(S_0)$. Thus, the alternative definition suggests a single initial state which is less precise and therefore enables verification of fewer properties.

5.1.3. The Abstract Transition Relation

A definition that is similar to existential abstraction could work in the case where ACTL* must be preserved. Using the notation of Definition 5.2, the abstract transition relation can be defined by:

$$\widehat{R}(a, b) \;\Leftrightarrow\; R^{\exists\exists}(\gamma(a), \gamma(b)).$$

However, as for the other components of the abstract model, the abstract interpretation framework provides the means for a more precise definition. Next we present an abstract transition relation that is more precise. It is denoted by $\widehat{R^A}$ in order to emphasize that it preserves ACTL*.

$$\widehat{R^A}(a,b)) \;\Leftrightarrow\; b \in \{\alpha(Y) | Y \in min\{Y' | R^{\exists\exists}(\gamma(a), Y')\}\}.$$

The difference between $\widehat{R}$ and $\widehat{R^A}$ can be explained as follows. Given an abstract state a, consider all $Y' \subseteq S$ such that there is a transition from some state in $\gamma(a)$ to some state in Y' (i.e., $R^{\exists\exists}(\gamma(a), Y')$). Then, $\widehat{R}$ connects a to $\alpha(Y')$ for each of these Y'. On the other hand, $\widehat{R^A}$ connects a to all $\alpha(Y)$ which are minimal (by the inclusion order) among these Y'. Clearly, $\widehat{R^A}$ connects a to fewer states, which are more precise. Note also that minimal Y's are always singletons.

Example 5.3 *The following example shows the difference between $\widehat{R^A}$ and $\widehat{R}$. It also demonstrates the ability of $\widehat{R^A}$ to verify more properties than $\widehat{R}$.*
Let $M = (S, S_0, R, L)$ where

- $S = \{s_1, s_2, s_3\}$,
- $S_0 = \{s_1\}$,
- $R = \{(s_1, s_2), (s_1, s_3), (s_2, s_2), (s_3, s_2)\}$;
- $L(s_1) = \{p\}$, $L(s_2) = \{p, q\}$, $L(s_3) = \{\neg p, q\}$.

$\widehat{M}$ is defined by

- $\widehat{S} = \{a_1, a_{23}, \top\}$ *where*
  - $\gamma(a_1) = \{s_1\}$, $\gamma(a_{23}) = \{s_2, s_3\}$ *and* $\gamma(\top) = S$.
  - $\alpha(\{s_1\}) = a_1$, $\alpha(\{s_2\}) = \alpha(\{s_3\}) = \alpha(\{s_2, s_3\}) = a_{23}$, $\alpha(\{s_1, s_2\}) = \alpha(\{s_1, s_3\}) = \alpha(S) = \top$.
- $\widehat{S_0} = \{a_1\}$,
- $\widehat{L}(a_1) = \{p\}$, $\widehat{L}(a_{23}) = \{q\}$ *and* $\widehat{L}(\top) = \emptyset$.
- $\widehat{R^A} = \{(a_1, a_{23}), (a_{23}, a_{23}), (\top, a_{23})\}$ *and*
  $\widehat{R} = \widehat{R^A} \cup \{(a_1, \top), (a_{23}, \top)(\top, a_{23}), (\top, \top)\}$.

Suppose we would like to verify the property $\mathbf{AXAG}q$ *for M. When we check the property on the model defined by $\widehat{R}$, we find that it is false, and therefore we do not know whether it holds for M. However, if we check it on the model defined by $\widehat{R^A}$, since it is true for this model we can conclude that it is true for M as well.*

Lemma 5.4 *Let M be a model and $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R^A}, \widehat{L})$ be an abstract model for M. Then $M \preceq \widehat{M}$. Thus, for every ACTL* formula φ and for every abstract state $a \in \widehat{S}$, $\widehat{M}, a \models \varphi \Rightarrow M, \gamma(a) \models \varphi$.*

To show that $M \preceq \widehat{M}$, we define a simulation relation $H \subseteq S \times \widehat{S}$. In order to enable the simulation to relate abstract and concrete states we need to change the requirement on the state labeling: If $H(s,a)$ then $\widehat{L}(a) \subseteq L(s)$. The relation $H(s,a) \Leftrightarrow s \in \gamma(a)$ can now be shown to be a simulation preorder. For ACTL*, Theorem 2.7 still holds with the new definition of $\preceq$ and thus implies the lemma.

5.2. ABSTRACT MODEL PRESERVING ECTL*

Until now we have only been concerned with abstractions preserving ACTL*. In this section we show how to define an abstraction which preserves ECTL*. The abstract model is defined in such a way that if an ECTL* formula is true for that model then it is also true for the concrete model. In the next section we show how to combine the abstractions for ACTL* and ECTL* into one abstraction that weakly preserves all of CTL*. Recall that bisimulation also preserves full CTL*. However, bisimulation provides strong preservation and therefore usually allows less reduction in the abstract model.

The ECTL*-preserving abstract model is identical to the ACTL*-preserving model $\widehat{M}$ defined above, except that the transition relation $\widehat{R^A}$ is replaced by a different transition relation, $\widehat{R^E}$.

The following observation explains the difference between $\widehat{R^A}$ and $\widehat{R^E}$. In order to preserve ACTL*, the set of abstract transitions should represent each of the concrete transitions. Additional transitions are also allowed. The abstract model then includes every behavior of the concrete model. Hence, every ACTL* property true for the abstract model is also true for the concrete model.

On the other hand, in order to preserve ECTL*, the set of abstract transitions must include only representatives of concrete transitions and nothing else. Thus, any behavior of the abstract model appears also in the concrete model. As a result, every ECTL* property true for the abstract model is true for the concrete model as well.

We will therefore have an abstract transition from a to b only if for *every* state in $\gamma(a)$ there is a transition to some state in $\gamma(b)$ (i.e., $R^{\forall\exists}(\gamma(a),\gamma(b))$). However, as for $\widehat{R^A}$, we suggest a better definition that connects a to fewer abstract states, which are more precise:

$$\widehat{R^E}(a,b)) \Leftrightarrow b \in \{\alpha(Y) | Y \in min\{Y' | R^{\forall\exists}(\gamma(a),Y')\}\}.$$

As in the case of $\widehat{R^A}$ and $R^{\exists\exists}$, $\widehat{R^E}$ contains less transitions than $R^{\forall\exists}$. Still, it does not allow to prove more ECTL* properties.

Lemma 5.5 *Let M be a model and $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R^E}, \widehat{L})$ be an abstract model. Then $\widehat{M} \preceq M$. Thus, for every ECTL* formula φ, and for every abstract state $a \in \widehat{S}$, $\widehat{M}, a \models \varphi \Rightarrow M, \gamma(a) \models \varphi$.*

Here we define $H(a,s) \subseteq \widehat{S} \times S$. However, we relate exactly the same abstract and concrete states: $H(a,s) \Leftrightarrow s \in \gamma(a)$. As before we require that if s and a are related (here they are related by $H(a,s)$ rather than $H(s,a)$) then $\widehat{L}(a) \subseteq L(s)$. With these changes, Theorem 2.7 holds for ECTL* and thus the lemma holds.

Note that because of the minimality requirements in $\widehat{R^A}$, $\widehat{R^E}$ may not be included in $\widehat{R^A}$.

Example 5.4 *Consider the model of Example 5.3, in which the transition (s_3, s_2) is replaced by (s_3, s_1), resulting in a new transition relation R'. Then, $\widehat{R^A}' = \{(a_1, a_{23}), (a_{23}, a_{23}), (a_{23}, a_1), (\top, \top)\}$. On the other hand, $\widehat{R^E}' = \{(a_1, a_{23}), (a_{23}, \top), (\top, \top)\}$. Note that the transition $(a_{23}, \top) \in \widehat{R^E}'$ is not in $\widehat{R^A}'$ since $\top$ is not minimal in the set of states connected to a_{23}.*

Using the model with the $\widehat{R^E}'$ transition relation, we can verify for the concrete model the ECTL property* **EF EG** q.

5.3. ABSTRACT MODELS PRESERVING FULL CTL*

In order to (weakly) preserve full CTL*, we now define an abstract model with a *mixed* transition relation: $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R^A}, \widehat{R^E}, \widehat{L})$. This model has two types of paths: *A-paths*, defined along $\widehat{R^A}$ transitions, and *E-path*, defined along $\widehat{R^E}$ transitions. The semantics of CTL* with respect to this model differs from the semantics in Definition 2.3 only in item 3.

- $s \models \mathbf{A}\, f$ if and only if for every A-path π from s, $\pi \models f$.
- $s \models \mathbf{E}\, f$ if and only if there exists an E-path π from s such that $\pi \models f$.

Theorem 5.6 *Let M be a model and $\widehat{M} = (\widehat{S}, \widehat{S_0}, \widehat{R^A}, \widehat{R^E}, \widehat{L})$ be a mixed abstract model. Then for every CTL* formula φ and for every abstract state $a \in \widehat{S}$, $\widehat{M}, a \models \varphi \Rightarrow M, \gamma(a) \models \varphi$.*

This theorem can be proved by induction of the formula structure. It can also be proved based on a *mixed simulation* over M and $\widehat{M}$. For details see [22].

In [22] approximations were defined in the context of abstract interpretation. Similarly to the framework of data abstraction, approximations here allow different levels of precision. It has also been shown how an approximation can be extracted from the program text.

6. Related Work

Several works have applied data abstraction in order to reduce the state space. Wolper and Lovinfosse [60] characterize a class of *data-independent* systems in which the data values never affect the control flow of the computation. Therefore, the datapath can be abstracted away entirely. Van Aelten et al. [2] have discussed a method for simplifying the verification of synchronous processors by abstracting away the datapath. Abstracting the datapath using uninterpreted function symbols is very useful for verifying pipeline systems [7, 11, 39].

In this paper we present a methodology for automatic construction of an initial abstract model, based on atomic formulas extracted from the program text. The atomic formulas are similar to the *predicates* used for abstraction by Graf and Saidi [35]. However, predicates are used to generate an abstract model, while atomic formulas are used to construct an *abstraction mapping*.

The use of counterexamples to refine abstract models has been investigated by a number of researchers. The localization reduction by Kurshan [41] is an iterative technique in which both the initial abstraction and the counterexample-guided refinements are based on the *variable dependency graph*. The localization reduction either leaves a variable unchanged or replaces it by a nondeterministic assignment. A similar approach has been described by Balarin et al. in [3] and by Lind-Nielson and Andersen [45]. The method presented here, on the other hand, applies abstraction mapping that makes it possible to distinguish many degrees of abstraction for each variable.

Lind-Nielson and Andersen [45] also suggest a model checker that uses upper and lower approximations in order to handle all of CTL. Their approximation techniques avoid the need to recheck the entire model after each refinement, yet still guarantee completeness.

A number of other papers [42, 52, 53] have proposed abstraction refinement techniques for CTL model checking. However, these papers do not use counterexamples to refine the abstraction. The methods described in these papers are orthogonal to the techniques presented here and may be combined with them in order to achieve better performance. The technique proposed by Govindaraju and Dill [33] is a first step in this direction. The paper only handles safety properties and path counterexamples; it uses random choice to extend the counterexample it constructs.

Many abstraction techniques can be viewed as applications of the abstract interpretation framework [19, 59, 20]. Bjorner, Browne and Manna use abstract interpretation to automatically generate invariants for general infinite state systems [8]. Abstraction techniques for the μ-calculus have

been suggested in [21, 22, 46].

Abstraction techniques for *infinite state systems* have been proposed in [1, 5, 43, 48]. The *predicate abstraction* technique, suggested by Graf and Saidi [35], is also aimed at abstracting an infinite state system into a finite state system. Later, a number of optimization techniques were developed in [6, 26, 25]. Saidi and Shankar have integrated predicate abstraction into the PVS system, which could easily determine when to abstract and when to model check [58]. Variants of predicate abstraction have been used in the Bandera Project [28] and the SLAM project [4].

Colón and Uribe [18] have presented a way to generate finite state abstractions using a decision procedure. As in predicate abstraction, their abstraction is generated using abstract Boolean variables.

A number of researchers have modeled or verified industrial hardware systems using abstraction techniques [32, 34, 37, 38]. In many cases, their abstractions are generated manually and combined with theorem-proving techniques [56, 57]. Dingel and Filkorn have used data abstraction and assume-guarantee reasoning, combined with theorem-proving techniques, in order to verify infinite state systems [27]. Recently, McMillan has incorporated a new type of data abstraction, along with assume-guarantee reasoning and theorem-proving techniques, into his Cadence SMV system [50].

7. Conclusion

In this work, three notions of abstraction have been introduced. They are all based on the idea that in order to check a specific property, some of the system states are in fact indistinguishable and can be collapsed into an abstract state that represents them.

Several concepts are common to all of these abstractions. Even though they were introduced within the framework of one of the abstractions, they are applicable with some changes to the other notions:

- The abstractions are derived from a high-level description of the program.
- Since deriving precise abstraction is usually difficult, the notion of approximations that are easier to compute is introduced.
- Refinement is required in case the abstraction is too coarse to enable the verification or the falsification of a given property.
- The abstractions provide weak preservation. Most of the discussion in this paper has been devoted to the preservation of ACTL*. However, preservation of ECTL* and full CTL* can be defined for each of the abstractions.

References

1. P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Computer-Aided Verification*, July 1999.
2. F. Van Aelten, S. Liao, J. Allen, and S. Devadas. Automatic generation and verification of sufficient correctness properties for synchronous processors. In *International Conference of Computer-Aided Design*, 1992.
3. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, volume 697 of *LNCS*, pages 29–40, 1993.
4. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, 2001.
5. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Computer-Aided Verification*, July 1992.
6. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification*, June 1998.
7. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Formal Methods in Computer-Aided Design*, pages 369–386, 1998.
8. N. S. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
9. 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.
10. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
11. J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification*, volume 818 of *LNCS*, pages 68–80, 1994.
12. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.
13. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
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. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV '00)*, LNCS, Chicago, USA, July 2000.
17. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.
18. M. A. Colón and T. E. Uribe. Generating finite-state abstraction of reactive systems using decision procedures. In *Computer-Aided Verification*, pages 293–304, 1998.
19. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM Symposium of Programming Language*, pages 238–252, 1977.
20. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6:69–95, 1999.

21. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Technical University of Eindhoven, Eindhoven, The Netherlands, 1995.
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. D. Dams, O. Grumberg, and R. Gerth. Abstraction interpretation of reactive systems: The preservation of CTL*. In *Programming Concepts, Methods and Calculi (ProCoMet)*. North Holland, 1994.
24. Dennis Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven university, Holland, July 1996.
25. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proc. of the Sixteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2001.
26. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 160–171. Springer Verlag, July 1999.
27. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69, Liege, Belgium, July 1995. Springer Verlag.
28. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001.
29. E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Lecture Notes in Computer Science 85*, pages 169–181. Automata, Languages and Programming, July 1980.
30. 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.
31. E.A. Emerson and Chin Laung Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages, New Orleans, La.*, January 1985.
32. D.A. Fura, P.J. Windley, and A.K. Somani. Abstraction techniques for modeling real-world interface chips. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 267–281, Vancouver, Canada, August 1993. University of British Columbia, Springer Verlag, published 1994.
33. S. G. Govindaraju and D. L. Dill. Verification by approximate forward and backward reachability. In *Proceedings of International Conference on Computer-Aided Design*, November 1998.
34. S. Graf. Verification of distributed cache memory by using abstractions. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 207–219, Standford, California, USA, June 1994. Springer Verlag.
35. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *LNCS*, pages 72–83, June 1997.
36. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
37. P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *International Conference of Computer-Aided Design*, pages 529–536, 1998.
38. R. Hojati and R. K. Brayton. Automatic datapath abstraction in hardware systems. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 98–113, Liege, Belgium, July 1995. Springer Verlag.

39. R. B. Jones, J. U. Skakkebak, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Formal Methods in Computer-Aided Design*, pages 2–17, 1998.
40. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.
41. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.
42. W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *International Conference of Computer-Aided Design*, pages 76–81, 1996.
43. D. Lesens and H. Sadi. Automatic verification of parameterized networks of processes by abstraction. In *International Workshop on Verification of Infinite State Systems (INFINITY)*, Bologna, July 1997.
44. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. Association for Computing Machinery, January 1985.
45. J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *Computer-Aided Verification*, volume 1633 of *LNCS*, pages 316–327. Springer Verlag, 1999.
46. 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.
47. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning.* PhD thesis, Carnegie Mellon University, 1993.
48. Z. Manna, M. Colon, B. Finkbeiner, H. Sipma, and T. Uribe. Abstraction and modular verification of infinit-state reactive systems. In *Requirements Targeting Software and Systems Engineering (RTSE)*, 1998.
49. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.
50. K. L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods*, September 1999.
51. 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.
52. A. Pardo. *Automatic Abstraction Techniques for Formal Verification of Digital Systems.* PhD thesis, University of Colorado at Boulder, Dept. of Computer Science, August 1997.
53. A. Pardo and G.D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.
54. D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.
55. A. Pnueli. The Temporal Semantics of Concurrent Programs. *Theor.Comp.Science*, 13:45–60, 1981.
56. J. Rushby. 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*, pages 1–11, 1999.
57. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–192, 1999.
58. H. Saidi and N. Shankar. Abstract and model checking while you prove. In *Computer-Aided Verification*, number 1633 in LNCS, pages 443–454, July 1999.
59. J. Sifakis. Property preserving homomorphisms of transition systems. In *4th Workshop on Logics of Programs*, June 1983.
60. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with net-

work invariants. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, 1989.