

# Monotonic Abstraction-Refinement for CTL

Sharon Shoham and Orna Grumberg

Computer Science Department, Technion, Haifa, Israel,  
{sharonsh,orna}@cs.technion.ac.il

**Abstract.** The goal of this work is to improve the efficiency and effectiveness of the abstraction-refinement framework for CTL over the 3-valued semantics. We start by proposing a symbolic (BDD-based) approach for this framework. Next, we generalize the definition of abstract models in order to provide a *monotonic* abstraction-refinement framework. To do so, we introduce the notion of *hyper-transitions*. For a given set of abstract states, this results in a more precise abstract model in which more CTL formulae can be proved or disproved. We suggest an automatic construction of an initial abstract model and its successive refined models. We complete the framework by adjusting the BDD-based approach to the new monotonic framework. Thus, we obtain a monotonic, symbolic framework that is suitable for both verification and falsification of full CTL.

## 1 Introduction

The goal of this work is to improve the efficiency and effectiveness of the abstraction-refinement framework for CTL over the 3-valued semantics. We first suggest a symbolic (BDD-based) approach for this framework. Next, we generalize the definition of abstract models in order to provide a *monotonic* abstraction-refinement framework. The new definition results in more precise abstract models in which more CTL formulae can be proved or disproved. Finally, we adjust the BDD-based approach to the new monotonic framework.

Abstraction is one of the most successful techniques for fighting the state explosion problem in model checking [5]. Abstractions hide some of the details of the verified system, thus result in a smaller model. Usually, they are designed to be *conservative* for *true*, meaning that if a formula is true of the abstract model then it is also true of the concrete (precise) model of the system.

The branching-time logic CTL [5] is widely used in model checking. In the context of abstraction, often only the universal fragment of CTL, ACTL, is considered. Over-approximated abstract models are used for verification of ACTL formulae while under-approximated abstract models are used for their refutation.

Abstractions designed for full CTL have the advantage of handling both verification and refutation on the same abstract model. A greater advantage is obtained if CTL is interpreted w.r.t. the 3-valued semantics [11]. This semantics evaluates a formula to either *true*, *false* or *indefinite*. Abstract models can then be designed to be conservative for both *true* and *false*. Only if the value of a formula in the abstract model is indefinite, its value in the concrete model is unknown. In this case, a refinement is needed in order to make the abstract model more precise.

The first result of this paper is a BDD-based approach for this framework. We use a symbolic model checking for CTL with the 3-valued semantics [3]. If the model checking results in an indefinite value, we find a cause for this result and derive from it a criterion for refinement. Previous works [15, 18, 19] suggested abstraction-refinement mechanisms for various branching time logics over 2-valued semantics, for *specific* abstractions. In [20] the 3-valued semantics is considered. Yet, their abstraction-refinement is based on games and is not suitable for a symbolic evaluation.

In order to motivate our next result we need a more detailed description of abstract models for CTL. Typically, each state of an abstract model represents a set of states of the concrete model. In order to be conservative for CTL the abstract model should contain both *may* transitions ( $\xrightarrow{\text{may}}$ ) which over-approximate transitions of the concrete model, and *must* transitions ( $\xrightarrow{\text{must}}$ ), which under-approximate the concrete transitions [14, 8]. In our work we use abstract models which are called *Kripke Modal Transition Systems* (KMTS) [12, 10]. In KMTSs, for every abstract states  $s_a$  and  $s'_a$ ,  $s_a \xrightarrow{\text{may}} s'_a$  iff there *exists* a concrete state  $s_c$  represented by  $s_a$  and there *exists* a concrete state  $s'_c$  represented by  $s'_a$  such that  $s_c \rightarrow s'_c$  ( $\exists\exists$ -condition).  $s_a \xrightarrow{\text{must}} s'_a$  iff for *all*  $s_c$  represented by  $s_a$  there *exists*  $s'_c$  represented by  $s'_a$  such that  $s_c \rightarrow s'_c$  ( $\forall\exists$ -condition).

Refinements “split” abstract states so that the new, refined states represent smaller subsets of concrete states. Several abstraction-refinement frameworks have been suggested for ACTL and LTL with the 2-valued semantics, where abstractions are conservative for *true* [13, 4, 1, 6, 2]. There, the refined model obtained from splitting abstract states has less (may) transitions and is therefore *more precise* in the sense that it satisfies more properties of the concrete model. We call such a refinement *monotonic*.

For full CTL with the 3-valued semantics, an abstraction-refinement framework has been suggested in [20]. For such a framework, one would expect that after splitting, the number of must transitions will increase as the number of may transitions decreases. Unfortunately, this is not the case. Once a state  $s'_a$  is split, the  $\forall\exists$ -condition that allowed  $s_a \xrightarrow{\text{must}} s'_a$  might not hold any more. As a result, the refinement is not monotonic since CTL formulae that had a definite value in the unrefined model may become indefinite.

In [9] this problem has been addressed. They suggest to keep copies of the unrefined states in the refined model together with the refined ones. This avoids the loss of must transitions and guarantees monotonicity. Yet, this solution is not sufficient because the *old* information is still expressed w.r.t. the “unrefined” states and the *new* information (achieved by the refinement) is expressed w.r.t. the refined states. As a result the additional precision that the refinement provides cannot be combined with the old information. This is discussed extensively in Section 4.1.

In this work we suggest a different monotonic abstraction-refinement framework which overcomes this problem. For a given set of abstract states, our approach results in a more precise abstract model in which more CTL formulae have a definite value. Moreover, our approach avoids the need to hold copies of the unrefined states.

Inspired by [17], we define a *generalized KMTS* (GKMTS) in which must transitions are replaced by *must hyper-transitions*, which connect a single state  $s_a$  to a set of states  $A$ . A GKMTS includes  $s_a \xrightarrow{\text{must}} A$  iff for *all*  $s_c$  represented by  $s_a$  there *exists*  $s'_c$  represented by some  $s'_a \in A$  such that  $s_c \rightarrow s'_c$ . This weakens the  $\forall\exists$ -condition by allowing the resulting states  $s'_c$  to be “scattered” in several abstract states.

In general, the number of must hyper-transitions might be exponential in the number of states in the abstract model. In practice, optimizations can be applied in order to reduce their number. We suggest an automatic construction of an initial GKMTS and its successive refined models in a way that in many cases avoids the exponential blowup.

In order to complete our framework, we also adjust for GKMTSs the 3-valued symbolic model checking and the refinement suggested above for KMTSs. Thus, we obtain a monotonic, symbolic framework that is suitable for both verification and falsification of full CTL.

**Organization.** In Section 2 we give some background for abstractions and the 3-valued semantics. We also present a symbolic 3-valued model checking algorithm. In Section 3 we suggest a refinement mechanism that fits the symbolic 3-valued model checker. In Section 4 we present *generalized* KMTSs and their use as abstract models. Finally, we present our monotonic abstraction-refinement framework in Section 5.

## 2 Preliminaries

Let  $AP$  be a finite set of atomic propositions. In this paper we consider the logic CTL, defined as follows:  $\varphi ::= \text{tt} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid A\psi$  where  $p \in AP$ , and  $\psi$  is a *path formula* defined by  $\psi ::= X\varphi \mid \varphi U\varphi \mid \varphi V\varphi$ . Other operators can be expressed in the usual manner [5]. Let  $Lit = AP \cup \{\neg p : p \in AP\}$ . The (concrete) semantics of CTL formulae is defined w.r.t. a *Kripke structure*  $M = (S, S_0, \rightarrow, L)$ , where  $S$  is a finite set of states,  $S_0 \subseteq S$  is a set of initial states,  $\rightarrow \subseteq S \times S$  is a transition relation, which must be *total* and  $L : S \rightarrow 2^{Lit}$  is a labeling function, such that for every state  $s$  and every  $p \in AP$ ,  $p \in L(s)$  iff  $\neg p \notin L(s)$ . A *path* in  $M$  from  $s$  is an infinite sequence of states,  $\pi = s_0, s_1, \dots$  such that  $s = s_0$  and  $\forall i \geq 0, s_i \rightarrow s_{i+1}$ .

$[(M, s) \models \varphi] = \text{tt}$  (= ff) means that the CTL formula  $\varphi$  is true (false) in the state  $s$  of the Kripke structure  $M$ .  $[(M, \pi) \models \psi] = \text{tt}$  (= ff) has the same meaning for path formulae over paths (see [5]).  $M$  *satisfies*  $\varphi$ , denoted  $[M \models \varphi] = \text{tt}$ , if  $\forall s_0 \in S_0 : [(M, s_0) \models \varphi] = \text{tt}$ . Otherwise,  $M$  *refutes*  $\varphi$ , denoted  $[M \models \varphi] = \text{ff}$ .

### 2.1 Abstraction

We use *Kripke Modal Transition Systems* [12, 10] as abstract models that preserve CTL.

**Definition 1.** A *Kripke Modal Transition System (KMTS)* is a tuple  $M = (S, S_0, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L)$ , where  $S, S_0$  are defined as before,  $\xrightarrow{\text{must}} \subseteq S \times S$  and  $\xrightarrow{\text{may}} \subseteq S \times S$  are transition relations such that  $\xrightarrow{\text{may}}$  is total and  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$ <sup>1</sup>, and  $L : S \rightarrow 2^{Lit}$  is a labeling function such that for every state  $s$  and  $p \in AP$ , at most one of  $p$  and  $\neg p$  is in  $L(s)$ .

A finite or infinite sequence of states  $\pi = s_0, s_1, \dots$  is a *path* in  $M$  from  $s$  if  $s = s_0$  and for every two consecutive states  $s_i, s_{i+1}$  in the sequence,  $s_i \xrightarrow{\text{may}} s_{i+1}$ .  $\pi$  is a *must (may)* path if it is *maximal* and for every  $s_i, s_{i+1}$  we have that  $s_i \xrightarrow{\text{must}} s_{i+1}$  ( $s_i \xrightarrow{\text{may}} s_{i+1}$ ). The maximality is in the sense that  $\pi$  cannot be extended by any transition of the same type.

Note, that a Kripke structure can be viewed as a KMTS where  $\rightarrow = \xrightarrow{\text{must}} = \xrightarrow{\text{may}}$ , and for each state  $s$  and  $p \in AP$ , we have that exactly one of  $p$  and  $\neg p$  is in  $L(s)$ .

<sup>1</sup> The requirement that  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$  is not essential for the purposes of this paper.

**Construction of an Abstract KMTS.** Let  $M_C = (S_C, S_{0C}, \rightarrow, L_C)$  be a (concrete) Kripke structure. Let  $S_A$  be a set of *abstract states* and  $\gamma : S_A \rightarrow 2^{S_C}$  a total *concretization function* that maps each abstract state to the set of concrete states it represents.

An abstract model, in the form of a KMTS  $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$ , can then be defined as follows. The set of initial abstract states  $S_{0A}$  is built such that  $s_{0a} \in S_{0A}$  iff  $\exists s_{0c} \in S_{0C} : s_c \in \gamma(s_{0a})$ . The “if” is needed in order to preserve truth from  $M_A$  to  $M_C$ , while “only if” is needed to preserve falsity.

The labeling of an abstract state is defined in accord with the labeling of all the concrete states it represents. For  $l \in Lit : l \in L_A(s_a)$  only if  $\forall s_c \in \gamma(s_a) : l \in L_C(s_c)$ . It is thus possible that neither  $p$  nor  $\neg p$  are in  $L_A(s_a)$ . If the “only if” is replaced by “iff”, then we say that the abstract labeling function is *exact*.

The *may*-transitions in an abstract model are computed such that every concrete transition between two states is represented by them: if  $\exists s_c \in \gamma(s_a)$  and  $\exists s'_c \in \gamma(s'_a)$  such that  $s_c \rightarrow s'_c$ , then there exists a may transition  $s_a \xrightarrow{\text{may}} s'_a$ . Note that it is possible that there are additional may transitions as well. The *must*-transitions, on the other hand, represent concrete transitions that are common to all the concrete states that are represented by the source abstract state: a must transition  $s_a \xrightarrow{\text{must}} s'_a$  exists only if  $\forall s_c \in \gamma(s_a) \exists s'_c \in \gamma(s'_a)$  such that  $s_c \rightarrow s'_c$ . Note that it is possible that there are less must transitions than allowed by this rule. That is, the may and must transitions do not have to be *exact*, as long as they maintain these conditions.

Other constructions of abstract models can be used as well. For example, if  $\gamma$  is a part of a *Galois Connection* [7] ( $\gamma : S_A \rightarrow 2^{S_C}, \alpha : 2^{S_C} \rightarrow S_A$ ) from  $(2^{S_C}, \subseteq)$  to  $(S_A, \sqsubseteq)$ , then an abstract model can be constructed as described in [8] within the framework of *Abstract Interpretation* [7, 16, 8]. It is then not guaranteed that  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$ .

**3-Valued Semantics.** [12] defines the *3-valued semantics*  $[(M, s) \models^3 \varphi]$  of CTL over KMTSs, and similarly  $[(M, \pi) \models^3 \psi]$  for path formulae, preserving both satisfaction (tt) and refutation (ff) from the abstract to the concrete model. Yet, a new truth value,  $\perp$ , is introduced, meaning that the truth value over the concrete model is unknown and can be either tt or ff. Intuitively, in order to preserve CTL, we examine *truth* of a formula of the form  $A\psi$  along all the *may paths*. Its *falsity* is shown by a single *must path*.

**Definition 2 (Precision Preorder).** Let  $M_1, M_2$  be two KMTSs over states  $S_1, S_2$  and let  $s_1 \in S_1$  and  $s_2 \in S_2$ . We say that  $(M_1, s_1)$  is more precise than  $(M_2, s_2)$ , denoted  $(M_1, s_1) \leq_{CTL} (M_2, s_2)$ , if for every  $\varphi$  in CTL:  $[(M_2, s_2) \models^3 \varphi] \neq \perp \Rightarrow [(M_1, s_1) \models^3 \varphi] = [(M_2, s_2) \models^3 \varphi]$ . Similarly, we say that  $M_1$  is more precise than  $M_2$ , denoted  $M_1 \leq_{CTL} M_2$ , if for every  $\varphi$  in CTL:  $[M_2 \models^3 \varphi] \neq \perp \Rightarrow [M_1 \models^3 \varphi] = [M_2 \models^3 \varphi]$ .

The following definition formalizes the relation between two KMTSs that guarantees preservation of CTL formulae w.r.t. the 3-valued semantics.

**Definition 3 (Mixed Simulation).** [8, 10] Let  $M_1 = (S_1, S_{01}, \xrightarrow{\text{must}}_1, \xrightarrow{\text{may}}_1, L_1)$  and  $M_2 = (S_2, S_{02}, \xrightarrow{\text{must}}_2, \xrightarrow{\text{may}}_2, L_2)$  be two KMTSs. We say that  $H \subseteq S_1 \times S_2$  is a mixed simulation from  $M_1$  to  $M_2$  if  $(s_1, s_2) \in H$  implies the following:

1.  $L_2(s_2) \subseteq L_1(s_1)$ .
2. if  $s_1 \xrightarrow{\text{may}}_1 s'_1$ , then there is some  $s'_2 \in S_2$  s.t.  $s_2 \xrightarrow{\text{may}}_2 s'_2$  and  $(s'_1, s'_2) \in H$ .
3. if  $s_2 \xrightarrow{\text{must}}_2 s'_2$ , then there is some  $s'_1 \in S_1$  s.t.  $s_1 \xrightarrow{\text{must}}_1 s'_1$  and  $(s'_1, s'_2) \in H$ .

If there is a mixed simulation  $H$  such that  $\forall s_1 \in S_{01} \exists s_2 \in S_{02} : (s_1, s_2) \in H$ , and  $\forall s_2 \in S_{02} \exists s_1 \in S_{01} : (s_1, s_2) \in H$ , then  $M_2$  is greater by the mixed simulation relation than  $M_1$ , denoted  $M_1 \preceq M_2$ .

In particular, Definition 3 can be applied to a (concrete) Kripke structure  $M_C$  and an (abstract) KMTS  $M_A$  constructed based on  $S_A, \gamma$  as described above. By doing so, we get that  $M_A$  is *greater by the mixed simulation relation* than  $M_C$ . The mixed simulation  $H \subseteq S_C \times S_A$  can be induced by  $\gamma$  as follows:  $(s_c, s_a) \in H$  iff  $s_c \in \gamma(s_a)$ . Preservation of CTL formulae is then guaranteed by the following theorem.

**Theorem 1.** [10] *Let  $H \subseteq S_1 \times S_2$  be the mixed simulation relation from a KMTS  $M_1$  to a KMTS  $M_2$ . Then for every  $(s_1, s_2) \in H$  we have that  $(M_1, s_1) \leq_{ctl} (M_2, s_2)$ . We conclude that  $M_1 \leq_{ctl} M_2$ .*

Note that if the KMTS  $M$  is in fact a Kripke structure, then for every CTL formula we have that  $[(M, s) \stackrel{3}{\models} \varphi] = [(M, s) \models \varphi]$ . Therefore, Theorem 1 also describes the relation between the 3-valued semantics over an abstract KMTS and the concrete semantics over the corresponding concrete model.

**Exact KMTS.** If the labeling function and transitions of the constructed abstract model  $M_A$  are *exact*, then we get the *exact* abstract model. This model is *most precise* compared to all the KMTSs that are constructed as described above w.r.t. the given  $S_A, \gamma$ .

## 2.2 Symbolic 3-Valued Model Checking

[3] suggests a symbolic multi-valued model checking algorithm for CTL. We rephrase their algorithm for the special case of the 3-valued semantics, discussed in our work.

Let  $M$  be a KMTS and  $\varphi$  a CTL formula. For  $v \in \{\text{tt}, \text{ff}, \perp\}$  we denote by  $[\varphi]_v$  the set of states in  $M$  for which the truth value of  $\varphi$  is  $v$ . That is,  $s \in [\varphi]_v$  iff  $[(M, s) \stackrel{3}{\models} \varphi] = v$ . Model checking is done by computing these sets for the desired property  $\varphi$ . If all the initial states of  $M$  are in  $[\varphi]_{\text{tt}}$ , then  $[M \stackrel{3}{\models} \varphi] = \text{tt}$ . If at least one initial state is in  $[\varphi]_{\text{ff}}$ , then  $[M \stackrel{3}{\models} \varphi] = \text{ff}$ , and otherwise  $[M \stackrel{3}{\models} \varphi] = \perp$ .

The algorithm that computes the sets  $[\varphi]_{\text{tt}}$  and  $[\varphi]_{\text{ff}}$  uses the following notation. For  $Z \subseteq S : ax(Z) = \{s \mid \forall s' : s \xrightarrow{\text{may}} s' \Rightarrow Z(s')\}$  and  $ex(Z) = \{s \mid \exists s' : s \xrightarrow{\text{must}} s' \wedge Z(s')\}$ . The algorithm is as follows.

$$\begin{array}{ll}
[\text{tt}]_{\text{tt}} = S & [\text{tt}]_{\text{ff}} = \emptyset \\
[p]_{\text{tt}} = \{s \in S : p \in L(s)\} & [p]_{\text{ff}} = \{s \in S : \neg p \in L(s)\} \quad \text{for } p \in AP \\
[\neg\varphi_1]_{\text{tt}} = [\varphi_1]_{\text{ff}} & [\neg\varphi_1]_{\text{ff}} = [\varphi_1]_{\text{tt}} \\
[\varphi_1 \wedge \varphi_2]_{\text{tt}} = [\varphi_1]_{\text{tt}} \cap [\varphi_2]_{\text{tt}} & [\varphi_1 \wedge \varphi_2]_{\text{ff}} = [\varphi_1]_{\text{ff}} \cup [\varphi_2]_{\text{ff}} \\
[AX\varphi_1]_{\text{tt}} = ax([\varphi_1]_{\text{tt}}) & [AX\varphi_1]_{\text{ff}} = ex([\varphi_1]_{\text{ff}}) \\
[A(\varphi_1 U \varphi_2)]_{\text{tt}} = \mu Z. [\varphi_2]_{\text{tt}} \cup ([\varphi_1]_{\text{tt}} \cap ax(Z)) & \\
[A(\varphi_1 U \varphi_2)]_{\text{ff}} = \nu Z. [\varphi_2]_{\text{ff}} \cap ([\varphi_1]_{\text{ff}} \cup ex(Z)) & \\
[A(\varphi_1 V \varphi_2)]_{\text{tt}} = \nu Z. [\varphi_2]_{\text{tt}} \cap ([\varphi_1]_{\text{tt}} \cup ax(Z)) & \\
[A(\varphi_1 V \varphi_2)]_{\text{ff}} = \mu Z. [\varphi_2]_{\text{ff}} \cup ([\varphi_1]_{\text{ff}} \cap ex(Z)) & 
\end{array}$$

Furthermore, for every CTL formula  $\varphi$ ,  $[\varphi]_{\perp}$  is computed as  $S \setminus ([\varphi]_{\text{tt}} \cup [\varphi]_{\text{ff}})$ .

The fixpoint operators  $\mu Z. \tau(Z)$  and  $\nu Z. \tau(Z)$  are computed as follows. For  $Z \subseteq S$  we define  $\tau^i(Z)$  to be the  $i$ th application of  $\tau$  to  $Z$ . Formally,  $\tau^0(Z) = Z$  and for every  $i > 0$ :  $\tau^{i+1}(Z) = \tau(\tau^i(Z))$ . Since the transformers ( $\tau$ 's) used in the fixpoint

definitions of  $AU$  and  $AV$  are monotonic and continuous (similarly to [5]), then they have a least fixpoint ( $\mu$ ) and a greatest fixpoint ( $\nu$ ) [21]. Furthermore,  $\mu Z.\tau(Z)$  can be computed by  $\bigcup_i \tau^i(\emptyset)$  and  $\nu Z.\tau(Z)$  can be computed by  $\bigcap_i \tau^i(S)$ .

### 3 3-Valued Refinement

Model checking of an abstract KMTS w.r.t. the 3-valued semantics may end with an indefinite result, raising the need for a refinement of the abstract model. In this section we suggest a refinement mechanism that fits the use of the symbolic 3-valued model checking algorithm presented above. This results in a symbolic 3-valued abstraction-refinement algorithm for CTL. The suggested refinement follows similar lines as the refinement of [20], where a game-based model checking was used.

We start with some definitions and observations regarding the symbolic 3-valued model checking algorithm. For  $\varphi \in \{A(\varphi_1 U \varphi_2), A(\varphi_1 V \varphi_2)\}$  and for  $v \in \{\text{tt}, \text{ff}\}$  we denote by  $\llbracket \varphi \rrbracket_v^i$  the set of states at the beginning of the  $i$ th iteration of the fixpoint computation of  $\llbracket \varphi \rrbracket_v$  ( $i \geq 0$ ). Furthermore, for  $\perp$ , we define  $\llbracket \varphi \rrbracket_{\perp}^i$  to be the set  $S \setminus (\llbracket \varphi \rrbracket_{\text{tt}}^i \cup \llbracket \varphi \rrbracket_{\text{ff}}^i)$ . Note that the sets  $\llbracket \varphi \rrbracket_{\perp}^i$  are *not* necessarily monotonic and that  $\llbracket \varphi \rrbracket_{\perp}$  is *not* computed by a fixpoint computation.

For every state  $s \in \llbracket \varphi \rrbracket_{\perp}$  we define  $en(s)$  to be the number of the iteration where  $s$  was first added to  $\llbracket \varphi \rrbracket_{\perp}^i$ . Note that  $\llbracket \varphi \rrbracket_{\perp}^0 = \emptyset$ , therefore  $en(s)$  is always  $\geq 1$ . Also note that  $en(s)$  can be computed from the intermediate results of the fixpoint computation when needed, without having to remember it for every state. We also have the following.

**Lemma 1.** *If  $s \in \llbracket \varphi \rrbracket_{\perp}$  then  $\forall i \geq en(s): s \in \llbracket \varphi \rrbracket_{\perp}^i$ . Furthermore, if  $\varphi = A(\varphi_1 U \varphi_2)$  then  $\forall i < en(s): s \in \llbracket \varphi \rrbracket_{\text{ff}}^i$  and if  $\varphi = A(\varphi_1 V \varphi_2)$  then  $\forall i < en(s): s \in \llbracket \varphi \rrbracket_{\text{tt}}^i$ .*

We now describe our refinement. As in most cases, our refinement consists of two parts. First, we choose a criterion that tells us how to split the abstract states. We then construct the refined abstract model, using the refined abstract state space.

Suppose the model checking result is  $\perp$  and refinement is needed. This means that there exists at least one initial state  $s_0$  for which the truth value of  $\varphi$  is  $\perp$ , i.e.  $s_0 \in \llbracket \varphi \rrbracket_{\perp}$ . Our goal is to find and eliminate at least one of the causes of the indefinite result. We first search for a *failure state*. This is a state  $s$  such that (1) the truth value of some subformula  $\varphi'$  of  $\varphi$  in  $s$  is  $\perp$ ; (2) the indefinite truth value of  $\varphi'$  in  $s$  *affects* the indefinite value of  $\varphi$  in  $s_0$ ; and (3) the indefinite value of  $\varphi'$  in  $s$  can be *discarded* by splitting  $s$ . The latter requirement means that the state  $s$  *itself* is responsible for introducing (some) uncertainty. The other requirements demand that this uncertainty is relevant to the model checking result. A failure state is found by applying the following recursive algorithm on  $s_0$  and  $\varphi$  (where  $s_0 \in \llbracket \varphi \rrbracket_{\perp}$ ).

Given a state  $s$  and a formula  $\varphi'$  s.t.  $s \in \llbracket \varphi' \rrbracket_{\perp}$ , algorithm `FindFailure` returns a failure state and either an atomic proposition or a may transition as the *cause* for failure.

**Algorithm** `FindFailure` ( $s, \varphi'$ )

- If  $\varphi' = p \in AP$ : return  $s$  and  $p$  as the cause.
- If  $\varphi' = \neg \varphi_1$ : call `FindFailure` on  $s$  and  $\varphi_1$  (we know that  $s \in \llbracket \varphi_1 \rrbracket_{\perp}$ ).
- If  $\varphi' = \varphi_1 \wedge \varphi_2$ : call `FindFailure` on  $s$  and  $\varphi_i$  for some  $i \in \{1, 2\}$  such that  $s \in \llbracket \varphi_i \rrbracket_{\perp}$  (such  $i$  must exist).

- If  $\varphi' = AX\varphi_1$ :
  - If there exists  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi_1 \rrbracket_{\text{ff}}$  then return  $s$  and  $s \xrightarrow{\text{may}} s_1$  as the cause.
  - Otherwise, call FindFailure on  $s_1$  and  $\varphi_1$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi_1 \rrbracket_{\perp}$  (such  $s_1$  must exist).
- If  $\varphi' = A(\varphi_1 U \varphi_2)$  or  $A(\varphi_1 V \varphi_2)$ :
  - If  $s \in \llbracket \varphi_2 \rrbracket_{\perp}$  then call FindFailure on  $s$  and  $\varphi_2$ .
  - Otherwise, if  $s \in \llbracket \varphi_1 \rrbracket_{\perp}$  then call FindFailure on  $s$  and  $\varphi_1$ .
  - Otherwise, if there exists  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi' \rrbracket_{\text{ff}}$  then return  $s$  and  $s \xrightarrow{\text{may}} s_1$  as the cause.
  - Otherwise, if there exists  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi' \rrbracket_{\perp}$  and  $en(s_1) < en(s)$  then call FindFailure on  $s_1$  and  $\varphi$ .
  - Otherwise, choose  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi' \rrbracket_{\perp}$  (and  $en(s_1) \geq en(s)$ ) and return  $s$  and  $s \xrightarrow{\text{may}} s_1$  as the cause (such  $s_1$  must exist).

Note that the order of the “if” statements in the algorithm determines the failure state returned by the algorithm. Different heuristics can be applied regarding their order.

**Theorem 2.** *The algorithm is well defined, meaning that all the possible cases are handled and the algorithm can always proceed. Furthermore, it always terminates.*

Intuitively, at every moment FindFailure looks for a reason for the indefinite value of the current formula  $\varphi'$  in the current state  $s$ . If  $s$  itself is not responsible for introducing the indefinite value, then the algorithm greedily continues with a state and formula that affect the indefinite value of  $\varphi'$  in  $s$ . This continues until a failure state is reached.

**Theorem 3.** *Let  $s$  be the failure state returned by FindFailure. Then the cause returned by the algorithm is either (1)  $p \in AP$  such that neither  $p$  nor  $\neg p$  label  $s$ ; or (2) an outgoing may transition of  $s$  which is not a must transition.*

In the first possibility described by Theorem 3, the labeling of  $s$  causes it to be in  $\llbracket p \rrbracket_{\perp}$ , thus it introduces an uncertainty and is considered the cause for failure. To demonstrate why the second case is viewed as a cause for failure, consider a formula  $A\varphi_1$  which is indefinite in a state  $s$ . If  $s$  has an outgoing may transition to a state  $s_1$  where the value of  $\varphi_1$  is ff, then  $s$  is considered a failure state with the may transition (which is *not* a must transition, by Theorem 3) being the cause. This is because changing the may transition to a must transition will make the value of  $AX\varphi_1$  in  $s$  definite (ff). Alternatively, if all such transitions are eliminated, it will also make the value of  $AX\varphi_1$  in  $s$  definite (tt).

A more complicated example of a may transition being the cause for the failure is when  $\varphi'$  is either  $A(\varphi_1 U \varphi_2)$  or  $A(\varphi_1 V \varphi_2)$  and (1)  $s \notin \llbracket \varphi_2 \rrbracket_{\perp}$ , (2)  $s \notin \llbracket \varphi_1 \rrbracket_{\perp}$ , (3) there is no  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi' \rrbracket_{\text{ff}}$  and (4) there is no  $s_1 \in S$  such that  $s \xrightarrow{\text{may}} s_1$  and  $s_1 \in \llbracket \varphi' \rrbracket_{\perp}$  and  $en(s_1) < en(s)$ . In this case the algorithm considers  $s$  to be a failure state and the cause is a may transition (which is *not* a must transition, by Theorem 3) to a state  $s_1$  such that  $s_1 \in \llbracket \varphi' \rrbracket_{\perp}$  and  $en(s_1) \geq en(s)$ . To understand why this is a failure state, we focus on the case where  $\varphi' = A(\varphi_1 U \varphi_2)$ . By Lemma 1, at iteration  $en(s)$  ( $\geq 1$ ),  $s$  moved from  $\llbracket \varphi' \rrbracket_{\text{ff}}^{en(s)-1}$  to  $\llbracket \varphi' \rrbracket_{\perp}^{en(s)}$ . By the description of the fixpoint computation of  $\llbracket \varphi' \rrbracket_{\text{ff}}$  we conclude that  $s \notin \llbracket \varphi_2 \rrbracket_{\text{ff}} \cap (\llbracket \varphi_1 \rrbracket_{\text{ff}} \cup \text{ex}(\llbracket \varphi' \rrbracket_{\text{ff}}^{en(s)-1}))$ .

Yet, by (1),  $s \notin \llbracket \varphi_2 \rrbracket_{\perp}$  and thus  $s \in \llbracket \varphi_2 \rrbracket_{\text{ff}}$  (otherwise it would be in  $\llbracket \varphi_2 \rrbracket_{\text{tt}}$  and thus also in  $\llbracket \varphi' \rrbracket_{\text{tt}}$ , in contradiction). Moreover, since  $s_1$  is not yet in  $\llbracket \varphi' \rrbracket_{\perp}^{en(s)-1}$ , then by Lemma 1 it must be at  $\llbracket \varphi' \rrbracket_{\text{ff}}^{en(s)-1}$  at that time. We consider  $s \xrightarrow{\text{may}} s_1$  to be the cause for failure because if it was a must transition rather than a may transition then  $s$  would be in the set  $ex(\llbracket \varphi' \rrbracket_{\text{ff}}^{en(s)-1})$  and therefore would remain in the set  $\llbracket \varphi' \rrbracket_{\text{ff}}^{en(s)}$  for at least one more iteration. Thus it would have a better chance to remain in the set  $\llbracket \varphi' \rrbracket_{\text{ff}}$  until fixpoint is reached, changing the indefinite value of  $\varphi'$  in  $s$  to a definite one.

Once we are given a failure state  $s$  and a corresponding cause for failure, we guide the refinement to discard the cause for failure in the hope for changing the model checking result to a definite one. This is done as in [20], where the failure information is analyzed and used to determine how the set of concrete states represented by  $s$  should be split. A criterion for splitting *all* abstract states can then be found by known techniques, depending on the abstraction used (e.g. [6, 4]).

Having defined the refinement, we now have a symbolic abstraction-refinement algorithm for CTL that uses the 3-valued semantics. In the next sections we will show how this algorithm can be improved, by using a new notion of abstract models.

## 4 Generalized Abstract Models

In this section we suggest the notion of a generalized KMTS and its use as an abstract model which preserves CTL. This notion allows better precision of the abstraction.

### 4.1 Motivation

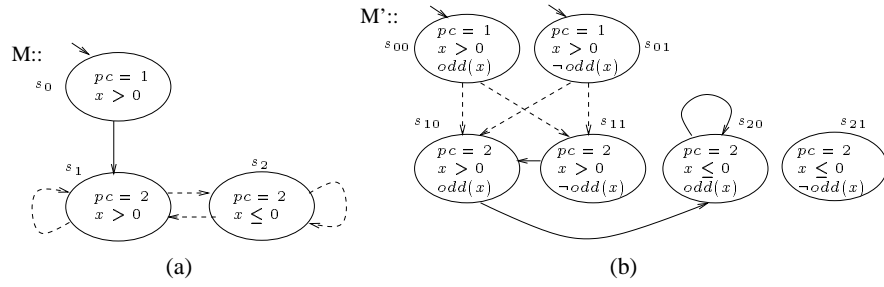
The main flaw of using KMTSs as abstract models is in the must transitions, which make the refinement not necessarily *monotonic* w.r.t. the precision preorder. The following example demonstrates the problem. We consider the traditional refinement that is based on splitting the states of the (abstract) model.

*Example 1.* Consider the following program  $P$ .

```
P:: input:  $x > 0$ 
   pc=1: if  $x > 5$  then  $x := x + 1$  else  $x := x + 2$  fi
   pc=2: while true do if odd( $x$ ) then  $x := -1$  else  $x := x + 1$  fi od
```

Suppose we are interested in checking the property  $\varphi = EF(x \leq 0)$ , which is clearly satisfied by this program. The concrete model of the program is an infinite state model. Suppose we start with an abstract model where concrete states that “agree” on the predicate  $(x \leq 0)$  (taken from the checked property  $\varphi$ ) are collapsed into a single abstract state. Then we get the abstract model  $M$  described in Fig. 1(a), where the truth value of  $\varphi$  is indefinite. Now, suppose we refine the model by adding the predicate *odd*( $x$ ). Then we get the model  $M'$  described in Fig. 1(b), where we still cannot verify  $\varphi$ . Moreover, we “lose” the must transition  $s_0 \xrightarrow{\text{must}} s_1$  of  $M$ . This transition has no corresponding must transition in the refined model  $M'$ . This loss causes the formula  $EX(x > 0)$  which is true in  $M$  to become indefinite in  $M'$ . Thus  $M' \not\prec_{\text{CTL}} M$ .

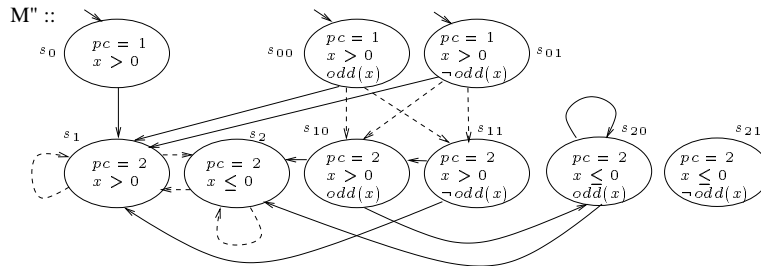
The source of the problem is that when dealing with KMTSs as abstract models, we are *not* guaranteed to have a mixed simulation between the refined abstract model and



**Fig. 1.** (a) An abstract model  $M$  describing the program  $P$ ; (b) The abstract model  $M'$  resulting from its refinement. Outgoing transitions of  $s_{21}$  are omitted since they are irrelevant.

the unrefined one, even if both are exact. This means that the refined abstract model is not necessarily more precise than the unrefined one, even though each of its states represents less concrete states. This is again demonstrated by Example 1. There, both the initial states of  $M'$  cannot be matched with the (only) initial state  $s_0$  of  $M$  in a way that fulfills the requirements of mixed simulation. This is because  $s_0$  has an outgoing must transition whereas the initial states of  $M'$  have none. Consequently,  $M' \not\leq M$ .

[9] suggests a refinement where the refined model *is* smaller by the mixed simulation than the unrefined one. The solution there is basically to use both the new refined abstract states and the old (unrefined) abstract states. This is a way of overcoming the problem that the destination states of must transitions are being split, causing an undesired removal of such transitions. This indeed prevents the loss of precision. Yet, this solution is not sufficient, as demonstrated by the following example.



**Fig. 2.** The model  $M''$  achieved by applying refinement as suggested in [9] on  $M$  from Fig. 1(a). Outgoing transitions of  $s_{21}$  are omitted since they are irrelevant, and so are additional outgoing may transitions of the unrefined states (there are no additional outgoing must transitions for the unrefined states).

*Example 2.* Fig. 2 presents the refined model  $M''$  achieved by applying refinement as suggested in [9] on the model  $M$  from Fig. 1(a). Indeed, we now have a mixed simulation relation from the refined model  $M''$  to the unrefined model  $M$ , by simply matching each state with itself or with its super-state, and the loss of precision is prevented. In particular, the truth value of  $EX(x > 0)$  in  $M''$  (unlike  $M'$  from Fig. 1(b)) is tt, since

there are must transitions from the initial states of  $M''$  to the *old* unrefined state  $s_1$ . Yet, in order to verify the desired property  $\varphi = EF(x \leq 0)$ , we need a must transition to (at least one of) the *new* refined states  $s_{10}$  and  $s_{11}$  from which a state satisfying  $x \leq 0$  is definitely reachable (this information was added by the refinement). However, the  $\forall\exists$  condition is still *not* fulfilled between these states. As a result we cannot benefit from the additional precision that the refinement provides and  $\varphi$  is *still* indefinite.

This example demonstrates that even when using the refinement suggested in [9], must transitions may still be removed in the “refined” part of the model, containing the new refined states. As a result the additional precision that the refinement provides cannot necessarily be combined with the old information.

## 4.2 Generalized KMTSs

Having understood the problems that result from the use of must transitions in their current form, our goal here is to suggest an alternative that will allow to weaken the  $\forall\exists$  condition. Following the idea presented in [17] (in a slightly different context), we suggest the use of *hyper-transitions* to describe must transitions.

**Definition 4 (Hyper-Transition).** *Given a set of states  $S$ , a hyper-transition is a pair  $(s, A)$  where  $s \in S$  and  $A \subseteq S$  is a nonempty set of states. Alternatively, a hyper-transition from a state  $s$  is a nonempty set of (regular) transitions from  $s$ .*

A (regular) transition  $(s, s')$  can be viewed as a hyper-transition  $(s, A)$  where  $A = \{s'\}$ .

Recall that a (regular) must transition exists from  $s_a$  to  $s'_a$  in an abstract model only if *every* state represented by  $s_a$  has a (concrete) transition to *some* state represented by  $s'_a$ . The purpose of the generalization is to allow such a concrete transition to exist to *some* state represented by *some* (abstract) state in a set  $A_a$  (which plays the role of  $s'_a$ ). This can be achieved by using a hyper-transition. The hyper-transition will still perform as a must transition in the sense that it will represent at least one concrete transition of each concrete state represented by  $s_a$  (maintaining the  $\forall\exists$  meaning).

**Definition 5.** *A Generalized Kripke Modal Transition System (GKMTS)  $M = (S, S_0, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L)$  is a KMTS except that  $\xrightarrow{\text{must}} \subseteq S \times 2^S$  and for every  $(s, A) \in \xrightarrow{\text{must}}$  and  $s' \in A$ , we have that  $(s, s') \in \xrightarrow{\text{may}}$  holds. Alternatively, viewing a hyper-transition  $(s, A)$  as a set of (regular) transitions  $\{(s, s') : s' \in A\}$ , we require that  $(s, A) \subseteq \xrightarrow{\text{may}}$ .*

The latter requirement replaces the requirement that  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$  in a KMTS. A KMTS can be viewed as a GKMTS where every must hyper-transition is a regular transition.

As before, a *may path* in  $M$  is an *infinite path* in  $M$ . However, instead of a *must path* we now have a *must hyper-path*. To formally define it we use the following notation.

**Definition 6.** *Let  $\Pi$  be a set of paths, then  $\text{pref}_i(\Pi)$  denotes the set of all the prefixes of length  $i$  of the paths in  $\Pi$ .*

**Definition 7 (Must Hyper-Path).** *A must hyper-path from a state  $s$  is a nonempty set  $\Pi$  of paths from  $s$ , such that for every  $i \geq 0$ :*

$$\text{pref}_{i+1}(\Pi) = \bigcup_{\pi_i \in \text{pref}_i(\Pi)} \{\pi_i \cdot s : s \in A_{\pi_i}\}$$

where for  $\pi_i = s_0, s_1, \dots, s_i \in \text{pref}_i(\Pi)$ , the set  $A_{\pi_i} \in 2^S$  is either (1) the target set of some must hyper-transition  $(s_i, A_{\pi_i})$ , or (2) the empty set,  $\emptyset$ , if there is no must hyper-transition exiting  $s_i$ .

Recall that our intention is to use GKMTSSs as abstract models. Considering this goal, Definition 7 is aimed at maintaining the desired property that if there is a must hyper-path  $\Pi$  from the abstract state  $s_a$  then every concrete state represented by  $s_a$  has a corresponding concrete path, i.e. a path that is represented by *some* path in  $\Pi$ .

Note that a must hyper-path can include finite paths since  $A_{\pi_i}$  can be empty.

**3-Valued Semantics.** We generalize the 3-valued semantics of CTL for GKMTSSs. The semantics is defined similarly to the (regular) 3-valued semantics, except that the use of must paths is replaced by must hyper-paths. In addition, for a (path) formula  $\psi$  of the form  $X\varphi$ ,  $\varphi_1 U \varphi_2$ , or  $\varphi_1 V \varphi_2$  and a must hyper-path  $\Pi$ , we define

- $[(M, \Pi) \models^3 \psi] = \text{tt (ff)}$ , iff for every  $\pi \in \Pi$  we have that  $[(M, \pi) \models^3 \psi] = \text{tt (ff)}$
- Otherwise,  $[(M, \Pi) \models^3 \psi] = \perp$ .

Note that the (regular) 3-valued semantics handles finite must paths as well.

The notion of a mixed simulation relation, that guaranteed preservation of CTL formulae between two KMTSSs, is generalized as well when dealing with GKMTSSs.

**Definition 8 (Generalized Mixed Simulation).** Let  $M_1 = (S_1, S_{01}, \xrightarrow{\text{must}}_1, \xrightarrow{\text{may}}_1, L_1)$  and  $M_2 = (S_2, S_{02}, \xrightarrow{\text{must}}_2, \xrightarrow{\text{may}}_2, L_2)$ , be two GKMTSSs. We say that  $H \subseteq S_1 \times S_2$  is a generalized mixed simulation from  $M_1$  to  $M_2$  if  $(s_1, s_2) \in H$  implies the following:

1.  $L_2(s_2) \subseteq L_1(s_1)$ .
2. if  $s_1 \xrightarrow{\text{may}}_1 s'_1$ , then there is some  $s'_2 \in S_2$  s.t.  $s_2 \xrightarrow{\text{may}}_2 s'_2$  and  $(s'_1, s'_2) \in H$ .
3. if  $s_2 \xrightarrow{\text{must}}_2 A_2$ , then there is some  $A_1 \subseteq S_1$  s.t.  $s_1 \xrightarrow{\text{must}}_1 A_1$  and  $(A_1, A_2) \in \tilde{H}$ , where  $(A_1, A_2) \in \tilde{H} \Leftrightarrow \forall s'_1 \in A_1 \exists s'_2 \in A_2 : (s'_1, s'_2) \in H$ .

If there is a generalized mixed simulation  $H$  such that  $\forall s_1 \in S_{01} \exists s_2 \in S_{02} : (s_1, s_2) \in H$ , and  $\forall s_2 \in S_{02} \exists s_1 \in S_{01} : (s_1, s_2) \in H$ , then  $M_2$  is greater by the generalized mixed simulation relation than  $M_1$ , denoted  $M_1 \preceq M_2$ .

**Theorem 4.** Let  $H \subseteq S_1 \times S_2$  be a generalized mixed simulation relation from a GKMTSS  $M_1$  to a GKMTSS  $M_2$ . Then for every  $(s_1, s_2) \in H$  we have that  $(M_1, s_1) \leq_{\text{CTL}} (M_2, s_2)$ . We conclude that  $M_1 \leq_{\text{CTL}} M_2$ .

**Construction of an Abstract GKMTSS.** Given a concrete Kripke structure  $M_C$ , a set  $S_A$  of abstract states and a concretization function  $\gamma$ , an abstract GKMTSS  $M_A$  is constructed similarly to an abstract KMTSS with the following difference: a must hyper-transition  $s_a \xrightarrow{\text{must}} A_a$  exists only if  $\forall s_c \in \gamma(s_a) \exists s'_c \in \left( \bigcup_{s'_a \in A_a} \gamma(s'_a) \right) : s_c \rightarrow s'_c$ .

This construction assures us that  $M_C \preceq M_A$  w.r.t. the generalized mixed simulation. Therefore, Theorem 4 guarantees preservation of CTL from  $M_A$  to  $M_C$ .

The use of GKMTSSs allows construction of abstract models that are more precise than abstract models described as KMTSSs, when using the same abstract state space and the same concretization function. This is demonstrated by the following example.

*Example 3.* Consider the *exact* KMTSS  $M$  described in Fig. 1(a) for the program  $P$  from Example 1. The state  $s_1$  has no outgoing must transition. Therefore, even verification of the simple formula  $EXEX(\text{true})$  fails, although this formula holds in every concrete model where the transition relation is total. Using a GKMTSS (rather than a KMTSS) as an abstract model allows us to have a must hyper-transition from  $s_1$  to the set  $\{s_1, s_2\}$ . Therefore we are now able to verify the tautological formula  $EXEX(\text{true})$ .

**Exact GKMTS.** As with KMTSs, the must hyper-transitions of a GKMTS do not have to be *exact*, as long as they maintain the new  $\forall\exists$  condition. That is, it is possible to have less must hyper-transitions than allowed by the  $\forall\exists$  rule. If all the components of the GKMTS are exact, then we get the *exact* GKMTS, which is *most precise* compared to all the GKMTSs that are constructed by the same rules based on the given  $S_A, \gamma$ .

Any abstract GKMTS and in particular the exact GKMTS can be reduced without damaging its precision, based on the following observation. Given two must hyper-transitions  $s_a \xrightarrow{\text{must}} A_a$  and  $s_a \xrightarrow{\text{must}} A'_a$ , where  $A_a \subset A'_a$ , the transition  $s_a \xrightarrow{\text{must}} A'_a$  can be discarded without sacrificing the precision of the GKMTS. Therefore, a possible optimization would be to use only *minimal* must hyper-transitions where  $A_a$  is minimal. This is similar to the approach of [8], where the destination state of a (regular) must transition is chosen to be the smallest state w.r.t. a given partial order on  $S_A$ .

In general, even when applying the suggested optimization, the number of must hyper-transitions in the exact GKMTS might be exponential in the number of states. In practice, computing *all* of them is computationally expensive and unreasonable. Later on, we will suggest how to choose an initial set of must hyper-transitions and increase it gradually in a way that in many cases avoids the exponential blowup.

## 5 Monotonic Abstraction-Refinement Framework

In this section our goal is to show how GKMTSs can be used in practice within an abstraction-refinement framework designed for full CTL. We also show that using the suggested framework allows us to achieve the important advantage of a *monotonic* refinement when dealing with full CTL and not just a universal fragment of it.

We start by pointing out that using *exact* GKMTSs as abstract models solves the problem of the non-monotonic refinement, described in Section 4.1.

**Definition 9 (Split).** Let  $S_C$  be a set of concrete states, let  $S_A$  and  $S'_A$  be two sets of abstract states and let  $\gamma : S_A \rightarrow 2^{S_C}$ ,  $\gamma' : S'_A \rightarrow 2^{S_C}$  be the corresponding concretization functions. We say that  $(S'_A, \gamma')$  is a split of  $(S_A, \gamma)$  iff there exists a (total) function  $\rho : S'_A \rightarrow S_A$  such that for every  $s_a \in S_A$ :  $\left( \bigcup_{\rho(s'_a)=s_a} \gamma'(s'_a) \right) = \gamma(s_a)$ .

**Theorem 5.** Let  $M_C$  be a (concrete) Kripke structure and let  $M_A, M'_A$  be two exact GKMTSs defined based on  $(S_A, \gamma)$ ,  $(S'_A, \gamma')$  respectively, such that  $M_C \preceq M_A$  and  $M_C \preceq M'_A$ . If  $(S'_A, \gamma')$  is a split of  $(S_A, \gamma)$ , then  $M'_A \preceq M_A$ .

Theorem 5 claims that for exact GKMTSs, refinement that is based on splitting abstract states is monotonic. This is true without the need to hold “copies” of the unrefined abstract states. Yet, as claimed before, constructing the exact GKMTS is not practical. Therefore, we suggest a compromise that fits well into the framework of abstraction-refinement. We show how to construct an initial abstract GKMTS and how to construct a refined abstract GKMTS (based on splitting abstract states). The construction is done in a way that is on the one hand computationally efficient and on the other hand maintains a monotonic refinement. The basic idea is as follows. In each iteration of the abstraction-refinement we first construct an abstract KMTS, including its may transitions and its (regular) must transitions. We then compute additional must *hyper-transitions* as described below.

**Construction of an Initial Abstract Model  $M_0$ :**

Given an initial set of abstract states  $S_0$  and a concretization function  $\gamma_0$ :

1. construct an abstract KMTS based on  $(S_0, \gamma_0)$ .
2. for every abstract state, add a must hyper-transition which is the set of all its outgoing may transitions.

Note that the set of all the outgoing may transitions of a state indeed fulfills the  $\forall\exists$  condition and thus can be added as a must hyper-transition. This results from the totality of the concrete transition relation along with the property that every concrete transition is represented by some may transition. We call such must hyper-transitions *trivial*.

**Construction of a Refined Model  $M_{i+1}$ :**

Suppose that model checking of the abstract model  $M_i$  resulted in an indefinite result and refinement is needed. Let  $(S_{i+1}, \gamma_{i+1})$  be the split of  $(S_i, \gamma_i)$ , computed by some kind of a refinement mechanism. Construct  $M_{i+1}$  as follows.

1. construct an abstract KMTS based on  $(S_{i+1}, \gamma_{i+1})$ .
2. for every must hyper-transition (including regular must transitions)  $s_i \xrightarrow{\text{must}} A_i$  in  $M_i$  and for every state  $s_{i+1} \in S_{i+1}$  that is a sub-state of  $s_i \in S_i$ , add to  $M_{i+1}$  the must hyper-transition  $\bigcup_{s'_i \in A_i} \{s_{i+1} \xrightarrow{\text{may}} s'_{i+1} : s'_{i+1} \text{ is a sub-state of } s'_i\}$ .
3. [optional] discard from  $M_{i+1}$  any must hyper-transition  $s_{i+1} \xrightarrow{\text{must}} A_{i+1}$  that is not *minimal*, which means that there is  $s_{i+1} \xrightarrow{\text{must}} A'_{i+1}$  in  $M_{i+1}$  where  $A'_{i+1} \subset A_{i+1}$ .

The purpose of step 2 above is to avoid the loss of information from the previous iteration, without paying an additional cost. To do so, we derive must hyper-transitions in  $M_{i+1}$  from must hyper-transitions in  $M_i$ , while avoiding the recomputation of the  $\forall\exists$  rule. Namely, if there is a must hyper-transition from  $s_i$  to  $A_i$  in  $M_i$ , then for every state  $s_{i+1}$  in  $M_{i+1}$  that is a sub-state of  $s_i$  we add an outgoing must hyper-transition to the set of all sub-states of states in  $A_i$ , excluding states to which  $s_{i+1}$  does not have a may transition. Clearly, given that  $s_i \xrightarrow{\text{must}} A_i$ , we are guaranteed that the  $\forall\exists$  condition holds for the corresponding hyper-transitions in  $M_{i+1}$  as well. Note that this is not damaged by excluding from the destination set states to which  $s_{i+1}$  does not have a may transition. This is because the lack of a may transition shows that the  $\exists\exists$  condition does not hold between  $s_{i+1}$  and the relevant states. Therefore they cannot contribute to the satisfaction of the  $\forall\exists$  condition anyway and can be removed. By using this scheme, the construction of the GKMTS requires no additional computational cost, compared to the construction of a (regular) KMTS.

The purpose of step 3 is to reduce the GKMTS without sacrificing its precision. Note that the reduction done in this step can be performed *during* step 2.

**Theorem 6.** *Let  $M_C$  be a concrete Kripke structure and let  $M_0, M_1, \dots, M_i, \dots$  be the abstract GKMTSs constructed as described above. Then*

- (1) for every  $i \geq 0$ :  $M_C \preceq M_i$ ; and (2) for every  $i \geq 0$ :  $M_{i+1} \preceq M_i$ .

Theorem 6 first ensures that the construction of the initial and the refined GKMTSs described above yields abstract models which are greater by the generalized mixed simulation relation than the concrete model. Moreover, it ensures that although we do not

use the exact GKMTSs, we still have a generalized mixed simulation relation between GKMTSs from different iterations in a monotonic fashion. This means that we do not lose information during the refinement and we get “closer” to the concrete model.

*Example 4.* To demonstrate these ideas we return to the program  $P$  from Example 1 and see how the use of GKMTSs as described above affects it. The initial GKMTS  $M_0$  is similar to the KMTS  $M$  from Fig. 1(a), with two additional *trivial* must hyper-transitions from  $s_1$  and from  $s_2$  to  $\{s_1, s_2\}$ . Yet, the truth value of  $\varphi = EF(x \leq 0)$  remains indefinite in this model. When we construct the refined model  $M_1$  (based on the addition of the predicate  $odd(x)$ ), we get a GKMTS that is similar to the KMTS  $M'$  from Fig. 1(b), but  $M_1$  also has additional must hyper-transitions. In particular, it has two trivial must hyper-transitions from both of its initial states to the set  $\{s_{10}, s_{11}\}$ . These must hyper-transitions are the refined version of the (regular) must transition from  $s_0$  to  $s_1$  in  $M_0$ : They exist because the initial states of  $M_1$  are sub-states of the initial state  $s_0$  of  $M_0$  and the set  $\{s_{10}, s_{11}\}$  consists of all the sub-states of  $s_1$ . Their existence in  $M_1$  allows to verify  $\varphi$ , since due to them each of the initial states now has an outgoing must hyper-path in which all the paths reach  $s_{20}$  where  $x \leq 0$ .

Example 4 also demonstrates our advantage over [9] which stems from the fact that our refinement does not use “copies” of the unrefined abstract states, unlike [9]. This example shows that in our approach the *old* information (from the unrefined model) is expressed with respect to the *new* refined states. Consequently, the old information and the new information, for which refinement was needed, can be combined, resulting in a better precision.

To conclude this section and make the suggested ideas complete, it remains to provide (1) a model checking algorithm that evaluates CTL formulae over GKMTSs, using the generalized 3-valued semantics; and (2) a suitable refinement mechanism to be applied when the model checking result is indefinite. Using these two components within the general framework suggested above, results in an actual abstraction-refinement framework where the refinement is monotonic.

**Model Checking.** As a model checking algorithm we suggest a simple generalization of the symbolic 3-valued algorithm presented in Section 2.2. The only change is in the definition of the operator  $ex(Z)$ , which is now defined to be

$$ex(Z) = \{s \mid \exists s'_1, \dots, s'_n : s \xrightarrow{\text{must}} \{s'_1, \dots, s'_n\} \wedge \bigwedge_{i=1}^n Z(s'_i)\}$$

**Refinement.** As for the refinement mechanism, we can use the algorithm suggested in Section 3 in order to find a failure state, analyze the failure and decide how to split the abstract states. To be convinced of that, one needs to notice that the refinement is based on may transitions only. Therefore no change is needed.

Moreover, the construction of a refined model  $M_{i+1}$  can be improved when this refinement mechanism is used. Namely, during the failure analysis it is possible to learn about additional must hyper-transitions that can be added to  $M_{i+1}$ . This is because if the cause for failure is a may transition  $s_i \xrightarrow{\text{may}} s'_i$  (in  $M_i$ ) then the split is done by separating the set  $S'_C$  of all the concrete states represented by  $s_i$  that have a corresponding outgoing transition, from the rest (see [20]). In this case, we are guaranteed that after the split,

the  $\forall\exists$  condition holds between the sub-state of  $s_i$  representing the concrete set  $S'_C$  and the set containing all the sub-states of  $s'_i$ . Therefore, we can add such a must hyper-transition to  $M_{i+1}$  without additional computational cost.

Other extensions of the refinement mechanism, which are more GKMTS-oriented and further exploit the use of must hyper-transitions, are omitted due to space limitations.

**Theorem 7.** *For finite concrete models, iterating the suggested abstraction-refinement process is guaranteed to terminate with a definite answer.*

## References

1. S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Computer-Aided Verification (CAV)*, Denmark, July 2002.
2. 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.
3. M. Chechik, B. Devereux, A. Gurfinkel, and S. Easterbrook. Multi-valued symbolic model-checking. Technical Report CSRG-448, University of Toronto, April 2002.
4. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, LNCS, Chicago, USA, July 2000.
5. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.
6. E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer Aided Verification (CAV)*, 2002.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *popl4*, pages 238–252, 1977.
8. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.
9. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, 2001.
10. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Computer Aided Verification (CAV)*, LNCS, Copenhagen, Denmark, July 2002.
11. P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, LNCS, January 2003.
12. M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. *LNCS*, 2028:155–169, 2001.
13. R.P. Kurshan. *Computer-Aided-Verification of Coordinating Processes*. Princeton University Press, 1994.
14. K.G. Larsen and B. Thomsen. A modal process logic. In *LICS*, July 1988.
15. W. Lee, A. Pardo, J. Jang, G. D. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD*, pages 76–81, 1996.
16. 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*, 1995.
17. K. S. Namjoshi. Abstraction for branching time properties. In *CAV*, Boulder, CO, July 2003.
18. A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional mu-calculus model checking. In *Computer Aided Verification (CAV)*, pages 12–23, 1997.
19. A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.
20. S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. In *Computer Aided Verification*, Boulder, CO, July 2003.
21. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 1955.