2-Valued and 3-Valued Abstraction-Refinement in Model Checking

Orna GRUMBERG
Computer Science Department, Technion,
Haifa, Israel

Abstract.
This paper presents two frameworks for abstraction-refinement in model checking. The first is the *CounterExample-Guided Abstraction-Refinement* (CEGAR) which can verify universal fragments of temporal logics and is based on a 2-valued semantics of temporal logics . The other is the *Three-valued Abstraction-Refinement* (TVAR) and is based on a 3-valued semantics of these logics.

We also present an application of the 3-valued framework for fully automatic compositional model checking. Based on this and other successful applications of the 3-valued framework we conclude that the additional power it provides is worth the extra efforts of having non-standard definitions and algorithms.

Keywords.
Model checking, abstraction, 3-valued abstraction, refinement, CEGAR, TVAR, compositional model checking.

1. Introduction

In this paper we present two frameworks for abstraction-refinement in model checking, based on two different semantics for temporal logics: The 2-valued and the 3-valued semantics. We then show how to exploit the 3-valued framework for compositional model checking.

Model checking [10] is an efficient procedure for automatic system verification. Given a finite-state model of a system and a desired property, it checks whether the system satisfies the property. Model checking is widely used for hardware and for software verification. Its main limitation, however, is the *state explosion problem* which refers to its high memory requirements.

One of the most useful approaches to fighting the state explosion problem is abstraction. *Abstraction* hides some of the system details that are considered irrelevant for the checked property. An abstract model is then constructed in a way which guarantees *preservation* of the checked property. That is, from the truth value of the property on the abstract model we can conclude its truth value on the full (concrete) model of the system. The abstract model is usually significantly

smaller in size (i.e., number of states and transitions) than the concrete model, making the application of model checking to it more feasible[1].

It sometimes happen that too many system details are eliminated via abstraction and as a result, the property cannot be correctly evaluated on the abstract model. In that case, a *refinement* is needed in order to add more system details into the model, thus making it closer to the concrete model.

The most widely used abstraction-refinement framework is based on the 2-valued (standard) semantics of temporal logics. The abstract model in this case is an *over-approximation* of the concrete model, meaning that it has more behaviors (but still less states and transitions). Such abstractions preserve the *validity* of properties in the *universal* fragment of temporal logics, but do not preserve their falsity. Thus, it is possible that a property is refuted on the abstract model, but the counterexample which demonstrates a bad abstract behavior do not have a corresponding counterexample in the concrete model. Such counterexamples are called *spurious*. If spurious counterexamples are identified, a refinement is applied, in order to eliminate them. This framework is called *CounterExample-Guided Abstraction-Refinement (CEGAR)* [9].

A different framework for abstraction-refinement is based on the 3-valued semantics of temporal logics. The abstract model here both over-approximates and under-approximates the concrete model. It preserves both *validity and falsity* of *full* temporal logics. However, a property may also be evaluated to a third *indefinite* truth value (sometimes denoted $\perp$), which indicates that the abstraction is too coarse. In this case, refinement is aimed at eliminating the indefinite result. This framework is called *Three-Valued Abstraction-Refinement(TVAR)* [35,36].

The advantage of the 2-valued framework is that abstract models are of the same form as concrete models (Kripke structures). Thus, semantics with respect to abstract model is defined in the standard way. Further, standard model checking algorithms can be applied in order to check properties on these models.

In contrast, the 3-valued semantics of temporal logics is defined with respect to different types of models (KMTSs). The semantics is defined differently and special-purpose model checking is required. Yet, the main advantage of TVAR is its ability to both verify and falsify a significantly more expressive specification language (the full temporal logics rather than just their universal fragment). Further, as we show next, TVAR provides a different perspective on system modeling and checking and thus enabling to develop new verification methodologies.

Another useful approach for fighting the state explosion problem is *compositional model checking*. In compositional model checking one tries to verify parts of the system separately in order to avoid the construction of the entire system. To account for the dependencies between the components, the Assume-Guarantee (AG) paradigm [27,33] suggests how to verify one module based on an *assumption* about the behavior of its environment, where the environment consists of the other system modules. The environment is then verified, in order to *guarantee* that it actually satisfies the assumption. Many of the works on compositional model checking are based on the Assume-Guarantee (AG) paradigm and

[1] Abstraction can also transform an infinite-state system into a (small) finite abstract model.

on learning [13,2,1,5]. These works are all designed for universal safety properties with the exception of [16], which learns the full class of ω-regular languages[2].

In this paper we present an alternative solution to compositional model checking, based on the work in [38]. We exploit the power of the 3-valued abstraction in order to provide a fully automatic *compositional technique* that can determine the truth values of *full* temporal logics with respect to a given system. Our goal is to verify a system M composed of several components without ever fully constructing M. For that we view each component M_i of M as a 3-valued abstraction of M. If the checked property φ is evaluated to *True* or *False* on M_i then we can conclude that this is indeed the truth value of φ on M. If, however, φ is evaluated to the indefinite value on all components, then *only the parts of the components that are relevant to the indefinite result* are composed and checked. Thus the construction of the full model is avoided.

Other applications of 3-valued (and multi-valued) abstractions has been applied successfully in different contexts (see Conclusion for more details). We thus conclude that the additional power of the 3-valued framework is worth the extra efforts of having non-standard definitions and algorithms.

2-Valued Abstraction and Refinement

2. Basic Definitions

We start by presenting the logic that will be used for specification. We describe the logic *μ-calculus* [28], which is highly expressive. Widely used temporal logics such as LTL, CTL and CTL* [10] can be expressed within this logic.

The μ-calculus

Let AP be a finite set of atomic propositions and $\mathcal{V}$ a set of propositional variables. We define the set of literals over AP to be $Lit = AP \cup \{\neg p : p \in AP\}$. We identify $\neg\neg p$ with p. The logic *μ-calculus* [28] in *negation normal form* over AP is defined by:

$$\varphi \ ::= \ l \mid Z \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

where $l \in Lit$ and $Z \in \mathcal{V}$. Intuitively, $\Box$ stands for "all successors", and $\Diamond$ stands for "exists a successor". μ denotes the least fixpoint, whereas ν denotes the greatest fixpoint. We will also write η for either μ or ν. Let $\mathcal{L}_\mu$ denote the set of *closed* formulas generated by the above grammar, where the fixpoint quantifiers μ and ν are variable binders. We assume that formulas are well-named, i.e. no variable is bound more than once in any formula. Thus, every variable Z *identifies* a unique subformula $fp(Z) = \eta Z.\psi$ of φ, where the set $Sub(\varphi)$ of *subformulas* of φ is defined in the usual way.

[2] ω-regular languages can express both safety and liveness, but are still universal in essence, i.e., no existential properties can be described

A useful sub-logic of the μ-calculus is its *universal fragment*, denoted $\forall\mathcal{L}_\mu$, in which the operator $\Diamond$ is not allowed. Specification written in this fragment can describe a property that holds for *all* behaviors of the system, but cannot specify a property of a specific behavior. On the other hand, both safety and liveness properties can be described. Safety properties are usually described by means of greatest fixpoints while liveness properties are given by least fixpoints.

As an example of properties written in $\forall\mathcal{L}_\mu$, consider the safety property "along all execution paths of the system and in every state on the path, p holds", for atomic proposition p (writen in CTL [10] as AGp). This property can be written by the $\forall\mathcal{L}_\mu$ formula

$$\nu Z(p \wedge \Box Z).$$

The liveness property "Along every execution path eventually p holds" (written in CTL as AFp) will be written by the $\forall\mathcal{L}_\mu$ formula

$$\mu Z(p \vee \Box Z).$$

In the context of model checking, systems are typically modelled as *Kripke structures* [10]. For simplicity, we will assume Kripke structures with a *single* initial state. Most of the definitions and results described here are presented in the relevant works for Kripke structures with multiple initial states.

Definition 2.1 (Kripke structures) *A Kripke structure $M = (AP, S, s^0, R, L)$ is a tuple where AP is a finite set of atomic propositions, S is a finite set of states, $s^0 \in S$ is the initial state, $R \subseteq S \times S$ is a transition relation*[3]*, and $L : S \to 2^{Lit}$ is a labeling function, such that for every state s and every $p \in AP$, exactly one of p and $\neg p$ is in $L(s)$.*

The *concrete semantics* $[\![\varphi]\!]^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ over AP w.r.t. a Kripke structure $M = (AP, S, s^0, R, L)$ is a mapping from S to $\{\mathrm{tt}, \mathrm{ff}\}$. $[\![\varphi]\!]^M(s) = \mathrm{tt}$ $(= \mathrm{ff})$ means that the formula φ is true (false) in the state s of the Kripke structure M. If $[\![\varphi]\!]^M(s^0) = \mathrm{tt}$ $(= \mathrm{ff})$, we say that M satisfies (falsifies) φ, denoted $M \models \varphi$ $(M \not\models \varphi)$. For the full definition of the concrete semantics of μ-calculus, see [28,10].

Simulation Relation

Next, we define a preorder over Kripke structures, called the *simulation relation* [32]. Intuitively, if structure M_2 is greater by the simulation relation than structure M_1 then M_2 has more behaviors than M_1. Universal properties characterize *all* behaviors of the model, thus every universal property that holds on M_2 also holds on M_1.

[3] For temporal logics the transition relation is usually required to be total. This makes the definition of their semantics simpler and also simplifies the definition of simulation relations. For μ-calculus this requirement is not needed.

We will later show how to exploit the simulation relation in model checking: Instead of checking the concrete (full) model of a system we will check an abstract model that has more behaviors but less states and transitions. Thus, applying model checking to it is easier. By the preservation theorem for simulation presented below, we will be able to conclude that every property that is true on the abstract model is also true on the concrete model (the opposite, however, is not necessarily true).

Simulation between two models is checked state-wise: One state is smaller than another by the simulation relation if they are identically labeled and for every successor of the smaller state there is a corresponding successor of the greater one. Formally, let $M_1 = (AP, S_1, s_1^0, R_1, L_1)$ and $M_2 = (AP, S_2, s_2^0, R_2, L_2)$ be two Kripke structures over AP.

Definition 2.2 (Simulation relation) *A relation $H \subseteq S_1 \times S_2$ is a* simulation relation *[32] over M_1 and M_2 if the following conditions hold:*

1. *The initial states are related. That is, $H(s_1^0, s_2^0)$.*
2. *For every s_1, s_2 such that $H(s_1, s_2)$,*
   - *$L_1(s_1) = L_2(s_2)$ 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 $s_1 \leq s_2$ for $H(s_1, s_2)$. M_2 *simulates* M_1 (denoted $M_1 \leq M_2$) if there exists a simulation relation H over M_1 and M_2.

The following theorem states the preservation of $\forall\mathcal{L}_\mu$ formulas. Clearly, the same preservation holds also for LTL and for the universal fragments of CTL and CTL* [21].

Theorem 2.3 *[31] Let $M_1 \leq M_2$. Then for every $\forall\mathcal{L}_\mu$ formula φ with atomic propositions in AP, $M_2 \models \varphi$ implies $M_1 \models \varphi$. Further, if $s_1 \leq s_2$ then $[\![\varphi]\!]^{M_2}(s_2) = tt$ implies $[\![\varphi]\!]^{M_1}(s_1) = tt$.*

Abstraction Mappings

We consider abstractions that collapse sets of concrete states into single abstract states. Let M be a Kripke structure over a set of states S. Given a set of abstract states $\widehat{S}$, the *concretization function* $\gamma : \widehat{S} \to 2^S$ indicates, for each abstract state $\widehat{s}$, the set of concrete states represented by $\widehat{s}$.

Such abstractions can be described in the framework of *Abstract Interpretation* [14,31,15].

3. Abstract Models

In this section we define an abstraction which is suitable for reasoning about universal logics, such as $\forall\mathcal{L}_\mu$. The abstract model (Kripke structure) is defined with respect to a given concrete one. It is guaranteed by construction to be greater by the simulation relation than the concrete model, thus preservation of universal logics is guaranteed. In practice, however, the concrete model is too large to fit into

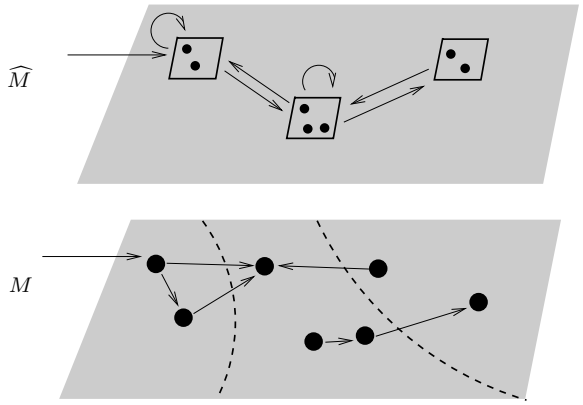

Figure 1. Existential Abstraction. M is the original Kripke structure, and $\widehat{M}$ the abstracted one. The dotted lines in M indicate how the states of M are clustered into abstract states.

memory and therefore is never produced. The abstract models are constructed directly from some high-level description of the system.

3.1. Existential Abstraction

We define abstract Kripke structures by means of *existential abstraction* [8]. Existential abstraction defines an abstract state to be an initial state if it represents an initial concrete state. Similarly, there is a transition from abstract state $\widehat{s}$ to abstract state $\widehat{s}'$ if there is a transition from a state represented by $\widehat{s}$ to a state represented by $\widehat{s}'$. An abstract model constructed by means of an existential abstraction is an *over-approximation* of the concrete model in the sense that every behavior of the concrete model has a corresponding behavior in the abstract model, but the abstract model may also contain additional behaviors. Formally,

Definition 3.1 (Abstract Kripke structure) *Let $M = (AP, S, S_0, R, L)$ be a (concrete) Kripke structure, let $\widehat{S}$ be a set of abstract states and $\gamma : \widehat{S} \to 2^S$ be a concretization function. The* abstract Kripke structure $\widehat{M} = (AP, \widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L})$ *generated by γ for M is defined as follows:*

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

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

Note that, an abstract state $\widehat{s}$ is labeled by a literal p if and only if all the states it represents are labeled by that literal. Since every concrete state is labeled

by exactly one of p and $\neg p$, $\widehat{s}$ will be labeled by *at most* one of p and $\neg p$. However, it might not be labeled by either of them. To avoid this we introduce an additional condition requiring that all concrete states, represented by the same abstract state, are identically labeled.

We say that a concretization function γ is *appropriate* for AP if for every abstract state $\widehat{s}$ and every $s_1, s_2 \in \gamma(\widehat{s})$, $L(s_1) = L(s_2)$. In that case we also obtain that $\widehat{L}(\widehat{s}) = L(s)$ for every $s \in \gamma(\widehat{s})$. With the appropriateness requirement, the resulting abstract model is a standard Kripke structure, thus standard semantics for the logic can be used and standard model checking algorithms can be applied to abstract models.

In section 5, we will consider 3-valued abstraction where the value of a literal on an abstract state may be *indefinite*.

As an example of a concrete and abstract models, consider a (part of a) model M described in the lower part of Figure 1. The dotted lines describe the partition of the concrete states into sets, each represented by an abstract state in $\widehat{M}$. The initial state and transitions in $\widehat{M}$ are defined according to the existential abstraction. Note that $\widehat{M}$ contains self loops indicating the possibility to stay forever in the same state. Such behaviors are not included in M. This exemplifies the fact that $\widehat{M}$ is an over-approximation of M.

The following theorem and corollary state that for $\forall\mathcal{L}_\mu$, properties which are correct for $\widehat{M}$ are correct for M as well.

Theorem 3.2 *[8] Let M be a Kripke structure and φ be a $\forall\mathcal{L}_\mu$ formula, both defined over AP. Further, let γ be appropriate for AP and $\widehat{M}$ be an abstract model generated by γ for M. Then $M \leq \widehat{M}$.*

By the above theorem and by Theorem 2.3 we get:

Corollary 3.3 *Let M and $\widehat{M}$ be as above. Then $[\![\varphi]\!]^{\widehat{M}} = tt$ implies $[\![\varphi]\!]^{M} = tt$.*

Note that once $\widehat{S}$, γ, and AP are given, $\widehat{S_0}$, $\widehat{R}$, and $\widehat{L}$ are uniquely determined. Thus, $\widehat{S}$, γ, and AP uniquely determine $\widehat{M}$. Since γ implicitly includes the information about $\widehat{S}$ and AP, we sometimes refer to γ for identifying $\widehat{M}$.

Several types of abstractions based on existential abstraction are defined and used. The most commonly used are the *localization reduction* [29] for hardware verification and *predicate abstraction* [19] for software verification. The abstractions differ in their choice of abstract states and the concretization function.

Localization reduction distinguishes between visible and invisible variables, where only the visible variables are considered to be relevant for the checked property. The abstract states are defined to be all valuation of the visible variables. γ mapped each abstract state to the set of concrete states that agree with it on the valuation of the visible variables (while they may differ in their valuation of the invisible variables). Usually, the visible variables are also chosen as the set of atomic propositions AP.

Predicate abstraction chooses a set of predicates over the program variables. Abstract states correspond to possible valuations of these predicates. An abstract

Figure 2. Abstraction of a US traffic light.

state represents all those concrete states which agree with it on the valuation of the predicates. The predicates are also used as the atomic propositions in AP.

As mentioned before, once $\widehat{S}$, γ, and AP are chosen, the abstract model defined by means of existential abstraction is uniquely determined. Other types of abstractions can also be found in the literature. A more detailed description of the abstractions mentioned above can be found in [22].

4. CounterExample-Guided Abstraction Refinement (CEGAR)

It is easy to see that, regardless of the type of abstraction we use, the abstract model $\widehat{M}$ contains less information than the concrete model M[4]. Thus, model checking the structure $\widehat{M}$ potentially produces incorrect results. Corollary 3.3 guarantees that if an $\forall\mathcal{L}_\mu$ specification is true in $\widehat{M}$ then it is also true in M. On the other hand, the following example shows that if the abstract model invalidates an $\forall\mathcal{L}_\mu$ specification, *the actual model may still satisfy the specification.*

Example 4.1 *The US traffic light controller M, presented on the left-hand side of Figure 2, contains three states, $red, green,$ and $yellow$. It is defined over the set of atomic propositions $AP = \{(state = red)\}$, where $L(red) = \{(state = red)\}$ and $L(green) = L(yellow) = \{\neg(state = red)\}$. We would like to prove for M the property "along every path, infinitely often $(state = red)$ holds". This can be written in CTL as $\psi = \mathbf{AG}\,\mathbf{AF}(state = red)$. The set of abstract states is $\{\widehat{red}, \widehat{go}\}$, where $\gamma(\widehat{red}) = \{red\}$ and $\gamma(\widehat{go}) = \{green, yellow\}$. Clearly, $\widehat{L}(\widehat{red}) = \{(state = red)\}$ while $\widehat{L}(\widehat{go}) = \{\neg(state = red)\}$. It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite abstract trace $\langle\widehat{red}, \widehat{go}, \widehat{go}, \dots\rangle$ that invalidates the property. However no corresponding concrete trace exists in M.*

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

Let us consider the situation outlined in Figure 3. We see that the abstract path, marked by the thick arrows, does not have a corresponding concrete path. Every concrete path starting at the initial state ends up in state D, from which we cannot go further. Therefore, D is called a *deadend state*. On the other hand, the state B is a *bad state*, since it made us believe that there is an outgoing transition. Finally, state I is an *irrelevant state* since it is neither deadend nor bad. To eliminate the spurious path, the abstraction can be refined, for instance,

[4]From now on we will assume that $\widehat{M}$ is defined according to a concretization function γ, which is appropriate for AP.

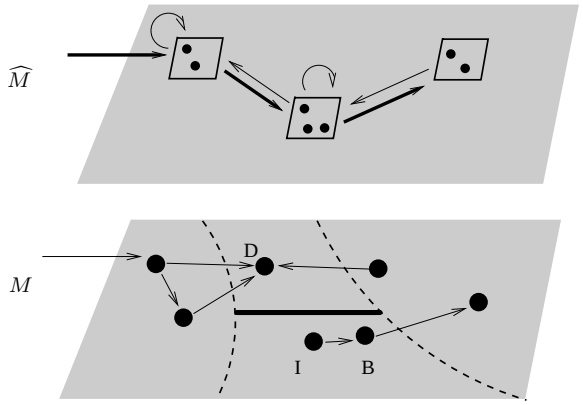

Figure 3. The abstract path in $\widehat{M}$ (indicated by the thick arrows) is spurious. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line in M.

as indicated by the thick line, separating deadend states from bad states. These notions are made precise in the next section.

4.1. The CEGAR Framework

In this section we present the framework of *CounterExample-Guided Abstraction-Refinement* (CEGAR) [29,9], for universal temporal logics and existential abstraction. The framework is suitable, in principle, for logics such as $\forall\mathcal{L}_\mu$, ACTL*, ACTL, and LTL. In practice, however, most model checking tools handle CTL or LTL. They usually produce a counterexample in the form of a finite path leading to a state violating the property. Alternatively, they produce a counterexample in the form of a lasso (finite path leading to a simple cycle), showing a behavior along which a desired state is never reached. Most CEGAR implementations refer to these forms of counterexamples.

The main steps of the CEGAR framework are presented below:

1. Given a system $\mathcal{P}$ (whose concrete model is M) and a universal temporal formula φ, generate an initial abstract model $\widehat{M}$.
   This step is typically done by examining a high level description of $\mathcal{P}$. For software, for instance, we may examine the program text and choose conditions used in control statements such as **if** and **while** as predicates. Additional predicates will come from the atomic formulas in φ.
2. Model check $\widehat{M}$ with respect to φ. If φ is true, then conclude that the concrete model satisfies the formula and stop. If a counterexample $\widehat{T}$ is found, check whether it is also a counterexample in the concrete model. If it is, conclude that the concrete model does not satisfy the formula and stop. Otherwise, the counterexample is spurious. Proceed to step 3.
3. Refine the abstract model, so that $\widehat{T}$ will not be included in the new, refined abstract model. Go back to step 2.
   Refinement is typically done by *partitioning* an abstract state. By this we mean that the set of concrete states, represented by the abstract state, is

partitioned. The refinement can be accelerated in the cost of faster increase of the abstract model if the criterion obtained for partitioning one abstract state (e.g. a new predicate) is used to partition all abstract states.

4.2. A BDD-based Implementation of CEGAR

To exemplify one possible implementation of CEGAR, we follow [9], and present a BDD-based implementation of the CEGAR framework. In this implementation the abstract models and the refinements are computed and represented symbolically, using BDDs [4]. Such an implementation is feasible only when the concrete model of the system under consideration is *finite* and only moderately large.

Other CEGAR implementations are based on SAT-solvers or theorem provers. Depending on the type of γ and the size of M, the initial abstract model (i.e., abstract initial states and abstract transitions) can be built using BDDs, SAT solvers or theorem provers. Similarly, the partitioning of abstract states, performed in the refinement, can be done using BDDs (e.g. as in [9]), SAT solvers (e.g. as in [6]), or linear programming and machine learning (e.g. as in [11]).

We use standard BDD-based symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the property φ. If it does, then by Corollary 3.3 we conclude that the original Kripke structure also satisfies φ. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the rest of this section, we will focus on counterexamples which are *finite paths*. In [9], counterexamples consisting of a finite path followed by a loop (lasso) are also considered. In [12], tree-like counterexamples for all of ACTL are considered.

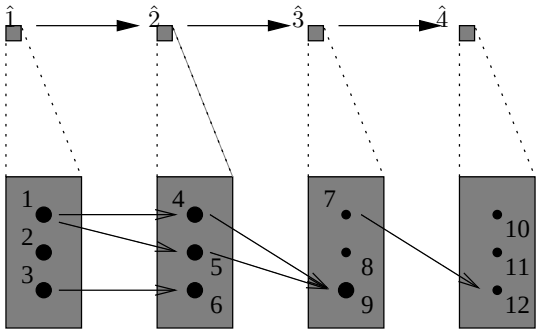

Figure 4. An abstract counterexample, which is spurious

4.2.1. Identifying Spurious Path Counterexamples

Assume the counterexample $\widehat{T}$ is a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$ starting at the initial abstract state $\widehat{s_1}$. We extend the concretization function γ to sequence of abstract states in the following way: $\gamma(\widehat{T})$ is the set of concrete paths defined as follows:

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

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

Lemma 4.1 *The following are equivalent:*

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

Suppose that condition (3) of Lemma 4.1 is violated, and let i be the largest index such that $S_i \neq \emptyset$. Then $\widehat{s_i}$ is called the *failure state* of the spurious counterexample $\widehat{T}$. It follows from Lemma 4.1 that if $\gamma(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal i ($1 \leq i \leq n$) such that $S_i = \emptyset$.

Example 4.2 *Consider a program with only one variable with domain $D = \{1, \cdots, 12\}$. Assume that the concretization function is shown in Figure 4 by the dotted lines from $\widehat{S} = \{\widehat{1}, \widehat{2}, \widehat{3}, \widehat{4}\}$. Suppose that we obtain an abstract counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 4.1, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that S_4 is empty. Thus, $\widehat{s_3}$ is the failure state.*

Algorithm SplitPATH$(\widehat{T})$

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

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

The symbolic Algorithm **SplitPATH** in Figure 5 obtains an abstract counterexample $\widehat{T} = \langle \widehat{s_1}, \ldots, \widehat{s_n} \rangle$ and computes the index of the failure state and the set of states S_{i-1}; the states in S_{i-1} are called *deadend* states. After the detection of the deadend states, we proceed to the refinement step. On the other hand, if none of the S_i sets is empty then **SplitPATH** will report that a "real" counterexample exists and we can stop.

4.2.2. Refining the Abstraction

We now explain how to refine an abstraction to eliminate a given spurious counterexample. In order to simplify the discussion we assume that the abstract model is exact (see the discussion following Definition 3.1). Abstract models with additional transitions and initial states can also be handled (see e.g. [26]). Recall the discussion concerning Figure 3 in Section 4 where we identified deadend states, bad states, and irrelevant states. The refinement should suggest a partitioning of abstract states, that will separate the deadend states S_D from the bad states S_B.

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

$$S_B = PreImage(\gamma(\widehat{s_{j+1}})) \cap \gamma(\widehat{s_j}),$$

where $PreImage$ computes the set of predecessors of the states in $\gamma(\widehat{s_{j+1}})$.

$\gamma(\widehat{s_j})$ should now be partitioned to separate between S_D and S_B. This can be done in different ways. For example, if we work directly with BDDs, then we can add a new abstract state $\widehat{s_j'}$ to $\widehat{S}$ and update the BDD for γ so that states in S_D are now mapped to the new state $\widehat{s_j}'$. Of course, now $\widehat{R}$, $\widehat{S_0}$ and $\widehat{L}$ should be updated.

Our refinement procedure continues to refine the abstraction by partitioning abstract states until a real counterexample is found, or the property is verified. Since we assume that the concrete model is finite, the partitioning procedure is guaranteed to terminate.

It should be noted that checking whether a counterexample is spurious and then finding a splitting criterion involves computations on the concrete structure. These computations, however, are usually easier than applying model checking to the concrete structure. This is because they refer to the part of it which is relevant to the counterexample. This is why BDD-based CEGAR is feasible only when the concrete model M is finite and only moderately large.

3-Valued Abstraction and Refinement

5. Abstract models and Mixed Simulation

In the previous sections we dealt with an abstraction which over-approximates the concrete model of the system. As seen there, such an abstraction can be used for verifying properties written in the universal fragment of the μ-calculus, but not for refuting such properties. In this section we present a different type of abstraction, the *3-valued abstraction*, which can be used for both verification and falsification of the full μ-calculus.

Abstract models preserving the full μ-calculus need to have two types of transitions [30,15]: The concrete transitions are over-approximated by the *may* transitions and under-approximated by the *must* transitions. This is achieved by using *Kripke Modal Transition Systems* [25,17] as abstract models.

Definition 5.1 *A* Kripke Modal Transition System *(KMTS) is a tuple* $M = (AP, S, s_0, R^+, R^-, L)$*, where* S *is a finite set of states,* $s_0 \in S$ *is an initial state,* $R^+ \subseteq S \times S$ *and* $R^- \subseteq S \times S$ *are transition relations such that* $R^+ \subseteq R^-$*, and* $L : S \to 2^{Lit}$ *is a labeling function, s.t. for each state* s *and* $p \in AP$*,* at most *one of* p *and* $\neg p$ *is in* $L(s)$*.*

The *3-valued semantics* $[\![\varphi]\!]_3^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ w.r.t. a KMTS M is a mapping from S to $\{\mathrm{tt}, \mathrm{ff}, \bot\}$ [3,25]. The interesting cases in the definition of the 3-valued semantics are those of the literals and the modalities.

$$[\![l]\!]_3^M(s) = \quad \mathrm{tt} \text{ if } l \in L(s), \quad \mathrm{ff} \text{ if } \neg l \in L(s), \quad \bot \text{ otherwise.}$$

$$[\![\Box\psi]\!]_3^M(s) = \begin{cases} \mathrm{tt}, \text{ if } \forall t \in S, \text{ if } R^-(s,t) \text{ then } [\![\psi]\!]_3^M(t) = \mathrm{tt} \\ \mathrm{ff}, \text{ if } \exists t \in S \text{ s.t. } R^+(s,t) \text{ and } [\![\psi]\!]_3^M(t) = \mathrm{ff} \\ \bot, \text{otherwise} \end{cases}$$

and dually for $\Diamond\psi$ when exchanging tt and ff. The notations $M \models \varphi$ and $M \not\models \varphi$ are used for KMTSs as well. In addition, if $[\![\varphi]\!]_3^M(s^0) = \bot$, the value of φ in M is indefinite.

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

Definition 5.2 (Mixed Simulation) *[15,17] Let* $M_1 = (AP, S_1, s_1^0, R_1^+, R_1^-, L_1)$ *and* $M_2 = (AP, S_2, s_2^0, R_2^+, R_2^-, L_2)$ *be two KMTSs, both defined over* AP*.* $\widehat{H} \subseteq S_1 \times S_2$ *is a* mixed simulation *from* M_1 *to* M_2 *if* $\widehat{H}(s_1, s_2)$ *implies:*

1. $L_2(s_2) \subseteq L_1(s_1)$.
2. *if* $R_1^-(s_1, s_1')$*, then there is some* $s_2' \in S_2$ *such that* $R_2^-(s_2, s_2')$ *and* $\widehat{H}(s_1', s_2')$.
3. *if* $R_2^+(s_2, s_2')$*, then there is some* $s_1' \in S_1$ *such that* $R_1^+(s_1, s_1')$ *and* $\widehat{H}(s_1', s_2')$.

If there is a mixed simulation $\widehat{H}$ *such that* $(s_1^0, s_2^0) \in \widehat{H}$*, then* M_2 abstracts M_1*, denoted* $M_1 \preceq M_2$*.*

In particular, Definition 5.2 can be applied to a (concrete) Kripke structure M and an (abstract) KMTS $\widehat{M}$, by viewing the Kripke structure as a KMTS where $R^+ = R^- = R$. For a Kripke structure, the 3-valued semantics agrees with the concrete semantics. Thus, preservation of $\mathcal{L}_\mu$ formulas is guaranteed by the following theorem.

Theorem 5.3 *[17] Let* $\widehat{H} \subseteq S_1 \times S_2$ *be the mixed simulation relation from a KMTS* M_1 *to a KMTS* M_2*. Then for every* $\widehat{H}(s_1, s_2)$ *and every* $\varphi \in \mathcal{L}_\mu$ *we have that* $[\![\varphi]\!]_3^{M_2}(s_2) \neq \bot \;\Rightarrow\; [\![\varphi]\!]_3^{M_1}(s_1) = [\![\varphi]\!]_3^{M_2}(s_2)$*.*

Given a concrete model $M = (AP, S, s_0, R, L)$, an abstract model $\widehat{M}$ can then be defined as follows. $\widehat{s_0}$ is the initial abstract state iff $s_0 \in \gamma(\widehat{s_0})$. An abstract state

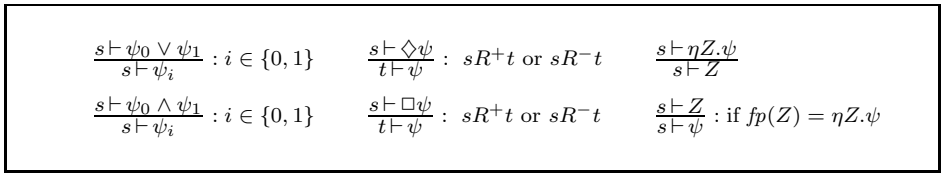

Figure 6. The rules of the model checking game for $\mathcal{L}_\mu$.

$\widehat{s}$ is labeled by $l \in Lit$ only if all the concrete states that it represents are labeled by l. Thus, it is possible that neither p nor $\neg p$ are in $\widehat{L}(\widehat{s})$. The *may*-transitions are computed such that they represent (at least) every concrete transition: if $\exists s \in \gamma(\widehat{s})$ and $\exists s' \in \gamma(\widehat{s}')$ such that $R(s,s')$, then $R^-(\widehat{s},\widehat{s}')$. The *must*-transitions represent concrete transitions that are common to all the concrete states represented by the origin abstract state: $R^+(\widehat{s},\widehat{s}')$ only if $\forall s \in \gamma(\widehat{s})\ \exists s' \in \gamma(\widehat{s}')$ such that $R(s,s')$. Other constructions of abstract models, based on Galois connections, can be found in [15,18].

The relation $\widehat{H} \in S \times \widehat{S}$, which is defined by $(s,\widehat{s}) \in \widehat{H}$ iff $s \in \gamma(\widehat{s})$, then forms a *mixed simulation* [15,17] from M to the resulting abstract model $\widehat{M}$. By the above theorem we thus have that every $\mathcal{L}_\mu$ formula which has a definite truth value (tt or ff) in the abstract KMTS $\widehat{M}$ has the same truth value in the concrete Kripke structure M as well.

6. 3-Valued Model Checking

A 3-valued game-based model checking for the μ-calculus over KMTSs was suggested in [20,23]. They introduce 3-valued parity games and translate the 3-valued model checking problem into the problem of determining the winner in a 3-valued satisfaction game, which is a special case of a 3-valued parity game. We omit the details of the 3-valued satisfaction game, but continue with the *game graph*, which presents all the information "relevant" for the model checking.

Let $M = (AP, S, s^0, R^+, R^-, L)$ be a KMTS and $\varphi \in \mathcal{L}_\mu$. The *game graph* $G_{M\times\varphi}$, or in short G, is a graph (N, n^0, E^+, E^-) where $N \subseteq S \times Sub(\varphi)$ is a set of nodes and $E^+ \subseteq E^- \subseteq N \times N$ are sets of must and may edges defined as follows. $n^0 = s^0 \vdash \varphi \in N$ is the initial node. The (rest of the) nodes and the edges are defined by the rules of Figure 6, with the meaning that whenever $n \in N$ is of the form of the upper part of the rule, the result in the lower part of the rule is also a node $n' \in N$ and $E^-(n,n')$. Moreover, $E^+(n,n')$ holds as well in all cases except for an application of the rules in the second column with a model's transition $(s,t) \in R^- \setminus R^+$. Intuitively, the outgoing edges of $s \vdash \psi \in N$ define "subgoals" for checking ψ in s.

If $E^-(n,n')$ ($E^+(n,n')$) then n' is a may (must) son of n. A node $n = s \vdash \psi$ in $G_{M\times\varphi}$ is classified as a $\wedge$, $\vee$, $\Box$, $\Diamond$ node, based on ψ. If ψ is of the form Z or $\eta Z.\psi'$, n is *deterministic* – it has exactly one son. If n has no outgoing edges, then it is a *terminal node*. In a full game graph this means that either ψ is a literal, or ψ is of the form $\Diamond\psi'$ or $\Box\psi'$, and s has no outgoing transition in M.

Figure 7(b) presents examples of game graphs for $\varphi = \Box(\neg i \vee \Diamond o)$ (written in CTL as $AX(\neg i \vee EXo)$) and the models from Figure 7(a), where all transitions are considered may transitions.

The model checking algorithms of [20,23] can be viewed as *coloring* algorithms that label (color) each node $n = s \vdash \psi$ in the game graph by T, F, ? depending on the truth value of ψ in the state s in M (based on the 3-valued semantics). The result of the coloring is a *3-valued coloring function* $\chi : N \to \{T, F, ?\}$.

The following formalizes the correctness of the coloring. For a (possibly not closed) formula ψ, ψ^* denotes the result of replacing every free occurrence of $Z \in \mathcal{V}$ in ψ by $fp(Z)$. Note that ψ^* is a closed formula. Further note that if ψ is closed, then $\psi^* = \psi$.

Definition 6.1 *Let $G_{M\times\varphi}$ be a game graph for a KMTS M and $\varphi \in \mathcal{L}_\mu$. A (possibly partial) coloring function $\chi : N \to \{T, F, ?\}$ for $G_{M\times\varphi}$ (or its subgraph) is* correct *if for every $s \vdash \psi \in N$, whenever $\chi(s \vdash \psi)$ is defined, then:*

1. $[\![\psi^*]\!]_3^M(s) = tt$ *iff* $\chi(s \vdash \psi) = T$.
2. $[\![\psi^*]\!]_3^M(s) = ff$ *iff* $\chi(s \vdash \psi) = F$.
3. $[\![\psi^*]\!]_3^M(s) = \perp$ *iff* $\chi(s \vdash \psi) = ?$.

Theorem 6.2 *[20,23] Let χ_F be the (total) coloring function returned by the coloring algorithm of [20] or [23] for $G_{M\times\varphi}$. Then χ_F is correct.*

The final coloring of the nodes reflects the 3-valued semantics of the logic: A $\wedge$-node or a $\Box$-node is colored T iff all its may sons are colored T (and in particular if it has no may sons), it is colored F iff it has a must son which is colored F, and otherwise it is colored ?. Dually for a $\vee$-node or a $\Diamond$-node when exchanging T and F. The color of $s \vdash l$ for $l \in Lit$ is T iff $l \in L(s)$, F iff $\neg l \in L(s)$, and ? otherwise. The result of the coloring is demonstrated in Figure 7(b).

Refinement

If the result of model checking on an abstract model is indefinite ($\perp$), a refinement is needed. When using the coloring algorithms of [20,23], an indefinite result is accompanied with a *failure state* and a *failure cause*. The failure cause is either a literal whose value in the failure state is $\perp$, or an outgoing may transition of the failure state in the underlying model which is not a must transition. Refinement is then performed by splitting the abstract states in a way that eliminates the failure cause (see [20,23]).

As an example, consider the game graph G_1 of Figure 7. The node $s_0 \vdash \neg i$ is colored by $\perp$ because the value of $\neg i$ in s_0 is indefinite. Thus, s_0 is a failure node and i is the cause. A refinement will split $\gamma(\widehat{s_0})$ to separate concrete states in which i is true from those in which $\neg i$ is true.

The state s_1 is also a failure state for G_1. This is because the node $s_1 \vdash \Diamond o$ is colored $\perp$. The reason for $\perp$ is the may transition going from s_1 to s_2, where o is true. Refinement will thus split the states in $\gamma(\widehat{s_1})$ to those from which there is a (concrete) transition to a state in $\gamma(\widehat{s_2})$ and those that do not have such a transition.

7. Compositional Model Checking and 3-Valued Abstraction Join Forces

In the coming sections we will present an application of 3-valued abstraction to compositional model checking, based on [38].

7.1. Partial Coloring and Subgraphs

In the following sections we use the game-based model checking in order to identify and focus on the places where the dependencies between components of the system affect the model checking result. In this section we set the basis for this, by investigating properties of the game graph and the coloring algorithms.

The coloring algorithms of [20,23] have the important property that they can be applied on a partially colored graph, in which case they extend the given coloring to the rest of the graph in a correct way. Moreover, the coloring can also be applied on a partially colored *subgraph*, and under certain assumptions it will yield a correct coloring of the subgraph. To formalize this, we need the following definitions.

Definition 7.1 *Let G be a game graph and χ_F its final coloring function. For a non-terminal node n in G we define its* witnessing sons *as follows, depending on its type:*

$\wedge, \Box$: *the witnessing sons are those colored F or ? by χ_F.*
$\vee, \Diamond$: *the witnessing sons are those colored T or ? by χ_F.*
deterministic: *the witnessing son is the only son.*

The sons are witnessing in the sense that they suffice to determine the color of the node, thus removing the rest of the node's sons from the graph does not change the result of the coloring. For example, if a $\wedge$-node or a $\Box$-node has no witnessing sons, meaning all its sons are colored T, then we know it should be colored T, and this is indeed how the coloring algorithms will color the node when keeping only the witnessing sons. Otherwise, the witnessing sons determine whether the node should be colored F or ?, thus one can correctly color the node by considering only them.

Definition 7.2 *A subgraph G' of a game graph G is* closed *if every node in G' is either a terminal node, or* all *its witnessing sons (and corresp. edges) from G are also in G'.*

Theorem 7.3 *Consider a closed subgraph G' of a game graph G with a partial coloring function χ which is correct and defined over (at least) all the terminal nodes in G'. Then applying the coloring algorithm of [20] or [23] on G' with χ as an initial coloring results in a correct coloring of G'.*

In fact, for the coloring of the subgraph to be correct, not *all* the witnessing sons are needed, as long as there is enough information to explain the correct coloring of each uncolored node. However, we will see that in our case we will need all of them, as we will deduce from the game graph of one component to

the game graph of the full system, where some of the nodes will be removed and for some an indefinite color (?) will change into T or F. This means that some of the witnessing sons will not remain witnessing sons in the game graph of the full system. Thus, we will not be able to know a-priori which of them is the "right" choice to include in a way that will also provide the necessary information for a correct coloring in the game graph of the full system.

Another notion that we will need later is the following.

Definition 7.4 (?-Subgraph) *Let G be a colored graph whose initial node is colored ?. The* ?-subgraph *is the least subgraph $G_?$ of G that obeys the following:*

- *the initial node is in $G_?$ (and is the initial node of $G_?$).*
- *For each node in $G_?$ which is colored ? in G all its witnessing sons (and corresponding edges) in G are included in $G_?$.*

$G_?$ is accompanied with a partial coloring function χ_I which is defined over the terminal nodes in $G_?$, and colors them as the coloring function χ_F of G.

The ?-subgraph $G_?$ and its initial coloring meet the conditions of Theorem 7.3. Intuitively, this means that $G_?$ contains *all* the information regarding the indefinite result. Figure 7(b) provides examples of ?-subgraphs.

7.2. Compositional Model Checking

In compositional model checking the goal is to verify a formula φ on a compound system without ever constructing its full compound model $M_1 \| M_2$. In our setting M_1 and M_2 are Kripke structures that synchronize on the joint labelling of the states. Since a Kripke structure is a special case of a KMTS where $R = R^+ = R^-$, we define the composition for the more general case of KMTSs. In the following we denote by Lit_1 and Lit_2 the sets of literals over AP_1 and AP_2, respectively

Definition 7.5 (Composable KMTSs) *Two KMTSs $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ are* composable *if their initial states agree on their joint labeling, i.e. $L_1(s_1^0) \cap Lit_2 = L_2(s_2^0) \cap Lit_1$.*

Definition 7.6 *Let $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ be two composable KMTSs. We define their composition, denoted $M_1 \| M_2$, to be the KMTS $(AP, S, s^0, R^+, R^-, L)$, where*

- $AP = AP_1 \cup AP_2$
- $S = \{(s_1, s_2) \in S_1 \times S_2 \mid L_1(s_1) \cap Lit_2 = L_2(s_2) \cap Lit_1\}$
- $s^0 = (s_1^0, s_2^0)$
- $R^+ = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^+ \text{ and } (s_2, t_2) \in R_2^+\}$
- $R^- = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^- \text{ and } (s_2, t_2) \in R_2^-\}$
- $L((s_1, s_2)) = L(s_1) \cup L(s_2)$

In particular, if M_1 and M_2 are Kripke structures with transition relations R_1 and R_2 respectively, then $M_1 \| M_2$ is a Kripke structure with $R = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1 \text{ and } (s_2, t_2) \in R_2\}$.

From now on we fix AP to be $AP_1 \cup AP_2$. For $i \in \{1,2\}$ we use $\overline{i}$ to denote the remaining index in $\{1,2\} \setminus \{i\}$.

We use the mechanism produced for abstractions of full branching time logics for the purpose of compositional verification. The basic idea is to view each Kripke structure M_i as a partial model that abstracts $M_1 \| M_2$.

Definition 7.7 *Let $M_i = (AP_i, S_i, s_i^0, R_i, L_i)$ be a Kripke structure. We lift M_i into a KMTS $M_i\uparrow = (AP, S_i, s_i^0, R_i^+\uparrow, R_i^-\uparrow, L_i\uparrow)$ over AP where $R_i^+\uparrow = \emptyset$, $R_i^-\uparrow = R_i$ and $L_i\uparrow\,(s) = L_i(s)$.*

That is, we view M_i as a KMTS $M_i\uparrow$ over AP (rather than AP_i). This immediately makes the value of each literal over $AP \setminus AP_i$ in each state of $M_i\uparrow$ indefinite (as neither p nor $\neg p$ are in $L_i(s)$) – indeed, it depends on $M_{\overline{i}}$. In addition, each transition of M_i is considered a may transition (since in the composition it might be removed if a matching transition does not exist in $M_{\overline{i}}$, but transitions can never be added).

Theorem 7.8 *$M_1 \| M_2 \preceq M_i\uparrow$. The mixed simulation is $\{((s_1,s_2), s_i) \mid (s_1,s_2) \in S\}$.*

Since each $M_i\uparrow$ abstracts $M_1 \| M_2$, we are able to first consider each component separately: Theorem 5.3 ensures that if φ has a definite value (tt or ff) in $M_i\uparrow$ under the 3-valued semantics, then the same value holds in $M_1 \| M_2$ as well. In particular, the values in $M_1\uparrow$ and $M_2\uparrow$ cannot be contradictory, and a definite value in one of them is suffices in order to determine the value in $M_1 \| M_2$.

The more typical case is that the value of φ on both $M_1\uparrow$ and $M_2\uparrow$ is indefinite. This reflects the fact that φ depends on both components and their synchronization. Typically, an indefinite result requires some refinement of the abstract model. In our case refinement means considering the composition with the other component. Still, in this case as well, having considered each component separately can guide us into focusing on the places where we indeed need to consider the composition of the components.

The game-based approach to model checking provides a convenient way for presenting this information. If the KMTS $M_i\uparrow$ is model checked using the algorithm of [20] or [23], then the result is a colored game graph, in which T and F represent definite results (i.e. truth values that hold no matter what the environment is), but the ? color needs to be resolved by considering the composition. This is where the ?-subgraph (see Definition 7.4) becomes handy, as it points out the places where this is really needed.

The ?-subgraph for each component is computed top-down, starting from the initial node. As long as a node colored ? is encountered, the search continues in a BFS manner by including the witnessing sons. Definite nodes which are included in the subgraph become terminal nodes, and their coloring defines the initial coloring function.

The ?-subgraphs of the two colored graphs present all the indefinite information that results from the dependencies between the components. Thus, to resolve the indefinite result, we compose the ?-subgraphs.

Definition 7.9 (Product Graph) *Let $G_{?1}$ and $G_{?2}$ be two ?-subgraphs as above with initial nodes $s_1^0 \vdash \varphi$ and $s_2^0 \vdash \varphi$ resp. We define their product to be the least graph $G_\| = (N_\|, n_\|^0, E_\|^+, E_\|^-)$ such that:*

- *$n_\|^0 = (s_1^0, s_2^0) \vdash \varphi$ is the initial node in $N_\|$.*
- *If $(s_1, s_2) \vdash \psi \in N_\|$ and $(s_1 \vdash \psi, s_1' \vdash \psi') \in E_1^-$ and $(s_2 \vdash \psi, s_2' \vdash \psi') \in E_2^-$ and $L_1(s_1') \cap Lit_2 = L_2(s_2') \cap Lit_1$ (i.e. (s_1', s_2') is a state of $M_1 \| M_2$), then: $(s_1', s_2') \vdash \psi' \in N_\|$ and $((s_1, s_2) \vdash \psi, (s_1', s_2') \vdash \psi')$ is in $E_\|^+$ and $E_\|^-$.*

Note that all the edges in $G_\|$ are must edges, whereas in the ?-subgraphs we had may edges (the transitions of each component were treated as may transitions in the lifted version). This is because the product graph already refers to the complete system $M_1 \| M_2$, where all transitions are concrete transitions (modeled as must transitions).

The product graph is constructed by a top-down traversal of the subgraphs, where, starting from the initial nodes, nodes that share the same formulas and whose states agree on the joint labeling are composed (recall that s_1^0 and s_2^0 agree on their joint labeling). Whenever two non-terminal nodes are composed, the outgoing edges are computed as the product of their outgoing edges, limited to legal nodes (w.r.t. the restriction to states that agree on their labeling). In particular, this means that if a node in one subgraph has no matching node in the other, then it will be omitted from the product graph. In addition, when a terminal node of one subgraph is composed with a non-terminal node of the other, the resulting node is a terminal node in $G_\|$.

We accompany $G_\|$ with an initial coloring function for its terminal nodes based on the initial coloring functions of the two subgraphs. We use the following observation:

Proposition 7.10 *Let $n = (s_1, s_2) \vdash \psi$ be a terminal node in $G_\|$. Then one of the following holds. Either (a) at least one of $s_1 \vdash \psi$ and $s_2 \vdash \psi$ is a terminal node in its subgraph, in which case at least one of them is colored with a definite color by the initial coloring of its subgraph, and contradictory definite colors are impossible. We denote this color by $col(n)$; Or (b) both $s_1 \vdash \psi$ and $s_2 \vdash \psi$ are non-terminal nodes but no outgoing edges were left in their composition.*

Definition 7.11 *We define the initial coloring function χ_I of $G_\|$ as follows. Let n be a terminal node in $N_\|$. If it fulfills case (a) of Prop. 7.10, then $\chi_I(n) = col(n)$. If it fulfills case (b), then $\chi_I(n) = T$ if n is a $\wedge$-node or a $\Box$-node, and $\chi_I(n) = F$ if n is a $\vee$-node or a $\Diamond$-node. χ_I is undefined for the rest of the nodes.*

In particular, if a terminal node in $G_\|$ results from a terminal node which is colored by ? in one subgraph and a terminal node which is colored by some definite color in the other, then the definite color takes over.

Note that the initial coloring function of the product graph colors all the terminal nodes by definite colors. Along with the property that all the edges in the product graph are must edges, this reflects the fact that the composition resolves all the indefinite information that existed in each component when it was considered separately. Therefore, when applying (one of) the coloring algorithm to

the product graph, all the nodes are colored by definite colors (in fact, a 2-valued coloring can be applied).

Theorem 7.12 *The resulting product graph $G_{\|}$ is a closed subgraph of the game graph over $M_1\|M_2$. In addition, the initial coloring function is correct w.r.t. $M_1\|M_2$ and defined over all the terminal nodes in the subgraph.*

By Theorem 7.3, this means that coloring $G_{\|}$ results in a correct result with respect to the model checking of φ in $M_1\|M_2$. Thus, to model check φ on $M_1\|M_2$ it remains to color the product graph $G_{\|}$. Note that the full graph for $M_1\|M_2$ is not constructed. To sum up, the algorithm is as follows.

Step 1 Model check each $M_i\uparrow$ separately (for $i \in \{1, 2\}$):

1. Construct the game graph G_i for φ and $M_i\uparrow$.
2. Apply the 3-valued coloring on G_i.
   Let χ_i be the resulting coloring function.

If $\chi_1(n_1^0)$ or $\chi_2(n_2^0)$ is definite, return the corresponding model checking result for $M_1\|M_2$.

Step 2 Consider the composition $M_1\|M_2$:

1. Construct the ?-subgraphs for G_1 and G_2.
2. Construct the product graph $G_{\|}$ of the ?-subgraphs.
3. Apply the 3-valued coloring on $G_{\|}$
   (with the initial coloring function).

Return the model checking result corresponding to $\chi_{\|}(n_{\|}^0)$.

Example 7.1 *Consider the components depicted in Figure 7(a). The atomic proposition o (short for output) is local to M_1, i (input) is local to M_2, and r (receive) is the only joint atomic proposition that M_1 and M_2 synchronize on. Suppose we wish to verify in $M_1\|M_2$ the property $\Box(\neg i \vee \Diamond o)$, which states that in all the successor states of the initial state, an input signal implies that there is a successor state where the output signal holds. Figure 7(b) depicts the colored game graph of each (lifted) component, and highlights the ?-subgraph of each of them. The product graph and its coloring is depicted in Figure 7(c), as an "intersection" of the two subgraphs. All the edges in the product graph are must edges. All nodes, and in particular the initial node, are colored T, thus the property is verified. One can see that most of the efforts were done on each component separately, and the product graph only considers a small part of the compound system.*

8. Conclusion

In this paper we describe two frameworks for abstraction-refinement in model checking. While having similar general flow of abstract-check-refine, the details of

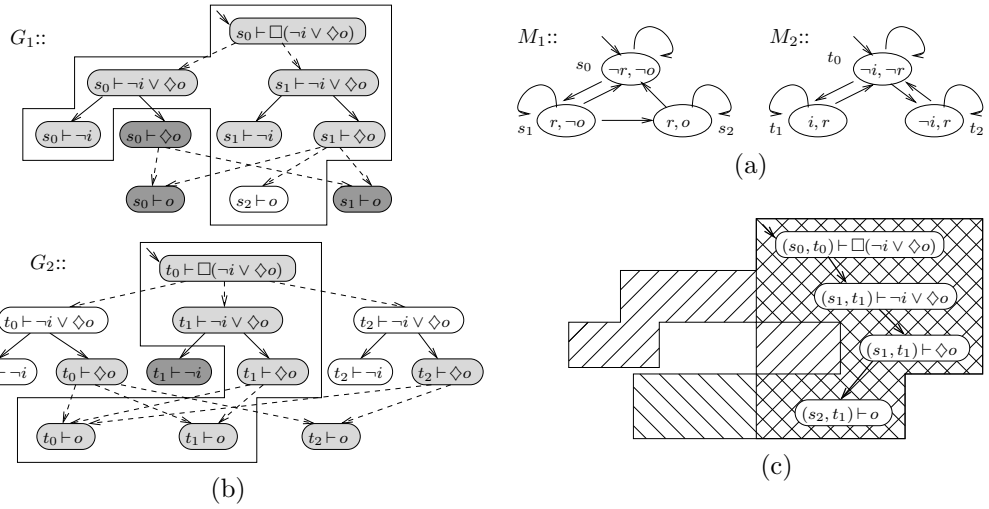

Figure 7. (a) Components, (b) their game graphs and their ?-subgraphs (enclosed by a line), and (c) the product graph. Dashed edges denote may edges which are not must edges. The colors reflect the coloring function: white stands for T, dark gray stands for F and light gray stands for ?.

the two frameworks are quite different. One is based on the 2-valued semantics for temporal logics. It may return false negative results since the abstraction does not distinguish between the case where the checked property is violated and the case where information is lacking. As a result, the goal of refinement is to eliminate spurious counterexamples.

The other framework is based on the 3-valued semantics. It identifies the case where information is lacking and returns then an *indefinite* result. It thus never returns false negative or false positive. The refinement in this case is aimed at eliminating indefinite results.

For a given set of abstract states, model checking the 3-valued abstract model will return more precise results more often. It is therefore worth the extra cost of constructing a KMTS rather than a Kripke structure as an abstract model. In [37] and [39], suggestions are made on how to reduce this extra cost while still benefit from the additional precision.

We demonstrate how the 3-valued abstraction-refinement framework can be exploited in order to obtain a fully automatic compositional technique. The compositional approach we present is significantly different from other compositional approaches, which are mostly based on the Assume-Guarantee reasoning.

Other applications of 3-valued abstractions have been applied successfully in different contexts. One example is the *Symbolic Trajectory Evaluation* (STE) [34] which combines symbolic simulation with 3-valued abstraction. It is capable of verifying and refuting huge circuits with respect to simple LTL specifications. However, it sometimes returns an indefinite value (denoted there by X) to indicate that the abstraction is too coarse to determine if the specification holds. In this case a refinement of the abstraction is applied [40,7] by replacing some of the X values on circuit inputs with symbolic variables.

X-BMC [41] is another successful application of 3-valued abstraction to hardware verification. It introduces X to the circuit inputs, thus eliminating parts of the circuit that are irrelevant to the checked property. It can handle any property expressed by an ω-regular language. An application of 3-valued abstraction to software is presented in [24].

All of the above demonstrates the power of 3-valued abstraction which can significantly enhance the model checking technology. Further research is needed in order to extend its use to other applications.

Acknowledgement: Sharon Shoham is thanked for significant contribution to the material which forms the basis for the second and third parts. Yael Meller is thanked for careful reading and useful comments.

References

[1] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *CAV*, 2005.

[2] H. Barringer, D. Giannakopoulou, and C.S. Pasareanu. Proof rules for automated compositional verification through learning. In *SAVCBS*, 2003.

[3] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*, pages 274–287, 1999.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986.

[5] Sagar Chaki, Edmund M. Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *CAV*, 2005.

[6] 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.

[7] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic STE refinement using responsibility. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, Budapest, Hungary, March 2008.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[9] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Journal of the ACM*, 50(5):752–794, 2003.

[10] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.

[11] E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine leraning techniques. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, July 2002.

[12] E.M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS)*, Copenhagen, Denmark, July 2002.

[13] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 331–346, Warsaw, Poland, April 2003.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252, Los Angeles, California, 1977.

[15] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.

[16] Azadeh Farzan, Yu-Fang Chen, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Extending automated compositional verification to the full class of omega-regular languages. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, Budapest, Hungary, March 2008.

[17] P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Proc. of Conference on Computer-Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 137–150, Copenhagen, Denmark, July 2002. Springer-Verlag.

[18] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR'01*, 2001.

[19] Sussanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. of Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.

[20] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. Don't know in the μ-calculus. In *VMCAI*, 2005.

[21] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[22] Orna Grumberg. Abstraction and refinement in model checking. In *International Conference on Formal Methods for Components and Objects (FMCO 2005)*, LNCS 4111, Leiden, The Netherlands, November 2005.

[23] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Compuatation*, 2007. doi: 10.1016/j.ic.2006.10.009.

[24] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, Vienna, Austria, April 2006.

[25] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. *Lecture Notes in Computer Science*, 2028:155–169, 2001.

[26] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27:366–379, February 2008.

[27] C.B. Jones. Specification and design of (parallel) programs. In *Information Processing 83: Proc. of the IFIP 9th World Congress*, pages 321–332. North-Holland, 1983.

[28] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.

[29] R. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[30] K.G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of Third Annual Symposium on Logic in Computer Science (LICS)*, pages 203–210. IEEE Computer Society Press, July 1988.

[31] 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.

[32] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489. BCS, 1971.

[33] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*. Springer-Verlag, 1984.

[34] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.

[35] Sharon Shoham. A game-based framework for CTL counterexamples and abstraction-refinement. Master's thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2003.

[36] Sharon Shoham. *Abstraction-Refinement and Modularity in μ-Calculus Model Checking*. PhD thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2009.

[37] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for ctl. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 331–346, Barcelona, April 2004.

[38] Sharon Shoham and Orna Grumberg. Compositional verification and 3-valued abstractions join forces. In *Static Analysis Symposium (SAS'07)*, volume 4634 of *LNCS*, Kongens Lyngby, Denmark, August 2007. Springer. To appear in Information and Computation.

[39] Sharon Shoham and Orna Grumberg. 3-valued abstraction: More precision at less cost. *Information and Computation*, 206(11):1313–1333, November 2008. A shorter version appeared in LICS 2006.

[40] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for Symbolic Trajectory Evaluation. In *18th International Conference on Computer Aided Verification (CAV'06)*, LNCS 4144, Seattle, August 2006.

[41] Avraham Yadgar. *New Approaches to Model Checking and 3-valued Abstraction and Refinement*. PhD thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2009.