A Game-Based Framework for CTL Counterexamples and 3-Valued Abstraction-Refinement

SHARON SHOHAM and ORNA GRUMBERG
Computer Science Department, Technion, Haifa, Israel

This work exploits and extends the game-based framework of CTL model checking for counterexample and incremental abstraction-refinement. We define a game-based CTL model checking for abstract models over the 3-valued semantics, which can be used for verification as well as refutation. The model checking process of an abstract model may end with an indefinite result, in which case we suggest a new notion of refinement, which eliminates indefinite results of the model checking. This provides an iterative abstraction-refinement framework. This framework is enhanced by an *incremental* algorithm, where refinement is applied only where indefinite results exist and definite results from prior iterations are used within the model checking algorithm. We also define the notion of *annotated counterexamples*, which are sufficient and minimal counterexamples for full CTL. We present an algorithm that uses the game board of the model checking game to derive an *annotated counterexample* in case the examined system model refutes the checked formula.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*; F.4.1 [**Logics and Meanings of Programs**]: Mathematical Logic—*Temporal Logic*

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Temporal logic, CTL, Model checking games, 3-valued semantics, Abstraction-Refinement, Counterexamples

1. INTRODUCTION

This work exploits and extends the game-based framework [Stirling 2001] of CTL model checking for counterexample and incremental abstraction-refinement.

Model checking [Clarke and Emerson 1981; Quielle and Sifakis 1982; Lichtenstein and Pnueli 1985] is a successful approach for verifying whether a system model M satisfies a specification φ, written as a temporal logic formula. Yet, concrete (regular) models of realistic systems tend to be very large, resulting in the *state explosion problem*. This raises the need for *abstraction*. Abstraction hides some of the system details, thus resulting in smaller models. Abstractions are usually designed to be *conservative* w.r.t. some logic of interest. That is, if the abstract model satisfies a formula in that logic then the concrete model satisfies it as well.

Authors' email address: {sharonsh,orna}@cs.technion.ac.il.

However, if the abstract model does not satisfy the formula then nothing is known about the concrete model.

In this work we refer to specifications written in the branching-time *computation-tree logic* (CTL) [Emerson and Clarke 1982]. Two types of semantics are available for interpreting CTL formulae over abstract models. The *2-valued* semantics defines a formula φ to be either true or false in an abstract model. True is guaranteed to hold for the concrete model as well, whereas false may be spurious. The *3-valued* semantics [Bruns and Godefroid 1999] introduces a new truth value: the value of a formula on an abstract model may be *indefinite*, which gives no information on its value on the concrete model. On the other hand, both satisfaction and falsification w.r.t. the 3-valued semantics hold for the concrete model as well. That is, while abstractions over 2-valued semantics are conservative w.r.t. only positive answers, abstractions over 3-valued semantics are conservative w.r.t. both positive and negative results. Abstractions over 3-valued semantics thus give precise results more often both for verification and falsification.

The use of abstraction is usually accompanied by *refinement*, for handling inconclusive results of model checking an abstract model. Such results are an indication that the abstraction cannot determine the value of the checked property in the concrete model and therefore needs to be refined. This leads to an iterative *abstraction-refinement* framework, where each iteration consists of (1) constructing an abstract model, (2) model checking the abstract model, and (3) refining the abstraction if the model checking result is inconclusive. The traditional abstraction-refinement framework [Kurshan 1994; Clarke et al. 2000] is designed for 2-valued abstractions, where false may be a false-alarm, thus refinement is aimed at eliminating false-alarms. As such, it is usually based on a counterexample analysis. Such an approach is not suitable for the 3-valued case, in which refinement is needed when the result is indefinite, rather than false. Refinement in this case should therefore be aimed at eliminating indefinite results and turning them into either definite true or definite false.

The refinement ingredient of an abstraction-refinement framework is still missing in current research on 3-valued abstractions. This is whereas the first two ingredients have already been investigated in the literature. Constructions of abstract models that preserve CTL were suggested, for example, in [Dams et al. 1997; Saidi and Shankar 1999; Godefroid et al. 2001]. Several model checking algorithms for CTL over the 3-valued semantics were suggested in [Bruns and Godefroid 1999; 2000; Chechik et al. 2001; Chechik et al. 2002; Godefroid and Jagadeesan 2003]. Yet, they lack an automatic refinement mechanism, which prevents them from achieving a complete abstraction-refinement framework.

Thus, the first goal of this work is to introduce an automatic refinement into the abstraction-refinement framework of CTL, while using the 3-valued semantics, where the abstract model can be used for both verification and falsification. An advantage of this work lies in the fact that the refinement is applied only to the indefinite part of the model. Thus, the refined abstract model does not grow unnecessarily. In addition, model checking of the refined model uses definite results from previous runs, resulting in an *incremental* model checking. Our abstraction-refinement process is complete in the sense that for a finite concrete model it will

always terminate with a definite "yes" or "no" answer.

The next goal of our work is to provide counterexamples for the full branching-time temporal logic CTL. When model checking a model M with respect to a property φ, if M does not satisfy φ then the model checker tries to return a counterexample. Typically, a counterexample is a part of the model that demonstrates the reason for the refutation of φ on M. Providing counterexamples is an important feature of model checking which helps tremendously in the debugging of the verified system.

Most existing model checking tools return as a counterexample either a finite path (for refuting formulae of the form AGp) or a finite path followed by a cycle (for refuting formulae of the form AFp) [1] [Clarke et al. 1995; Clarke et al. 1999]. Recently, this approach has been extended to provide counterexamples for all formulae of the universal fragment of an extended branching time logic based on ω-regular temporal operators, and in particular for ACTL [Clarke et al. 2002]. In this case the part of the model given as the counterexample has the form of a tree. This approach is still restricted to universal properties.

When full CTL is considered, we face existential properties as well. To prove refutation of an existential formula $E\psi$, one needs to show an initial state from which *all* paths do not satisfy ψ. Thus, the structure of the counterexample becomes more complex. Having such a complex counterexample, it might not be easy for the user to analyze it by looking at the subgraph of M alone.

This problem was dealt with in [Gurfinkel and Chechik 2003b]. There, CTL counterexamples are annotated with additional proof steps. Yet, they provide only limited information in the case of counterexamples for ECTL and properties that use both universal and existential quantifiers, since proof "obligations" are used. Their work is extended in [Gurfinkel and Chechik 2003a], where a multiple-valued extension of CTL is considered. Both these works use a model checker as a decision procedure in each step of the counterexample (proof) generation. As a result, they have the disadvantage of using multiple executions of model checking.

Our goal is to achieve a similar result as [Gurfinkel and Chechik 2003b; 2003a] while avoiding the overhead of multiple model checking runs. We propose an algorithm that uses the information gained in a *single* run of the model checker in order to construct a counterexample. Each state on the counterexample is *annotated* with a subformula of φ that is false in that state. The annotating subformulae being false in the respective states, provide the reason for φ to be false in the initial state. Thus, the annotated counterexample gives a convenient tool for debugging.

We prove that the counterexamples we produce are minimal and sufficient to explain refutation of any CTL formula. We also discuss several ways to use and present this information in practice.

Games for CTL model checking [Stirling 2001] are a most suitable framework for our goals. The model checking game is played by two players, $\forall$belard, the refuter who wants to show that $M \not\models \varphi$, and $\exists$loise, the prover who wants to show that $M \models \varphi$. The board of the game consists of pairs (s, ψ) of a model state and a subformula, with the meaning that the satisfaction of ψ in the state s is examined.

[1] AGp means "for every path, in every state on the path, p holds", whereas AFp means "along every path there is a state which satisfies p".

∀belard proceeds from such a node (s, ψ) to a node that helps refuting ψ on s. ∃loise chooses her moves with the intention to prove that s satisfies ψ. All possible plays of a game are captured in the *game-graph*, whose nodes are the elements of the game board and whose edges are the possible moves of the players. The initial nodes are pairs (s_0, φ) where s_0 is an initial state of M. It can be shown that ∀belard has a winning strategy (i.e., he can win the game regardless of ∃loise moves) iff $M \not\models \varphi$. ∃loise has a winning strategy iff $M \models \varphi$.

Model checking is then done by applying a *coloring algorithm* on the game-graph [Bollig et al. 2002]. It colors a node (s, ψ) by T iff ∃loise has a winning strategy, which means ψ is true in s. It colors it by F iff ∀belard has a winning strategy, which means ψ is false in s. At termination, if all initial nodes are colored T then $M \models \varphi$. If at least one initial node is colored F then $M \not\models \varphi$ and we would like to supply a counterexample.

In our work we add abstraction to the discussion. Concrete models for CTL are state-transition graphs (Kripke structures) in which nodes correspond to states of the system and transitions describe possible moves between states. Abstract models consist of abstract states, representing (not necessarily disjoint) sets of concrete states. In order to be conservative w.r.t. CTL, two types of transitions are required: *may*-transitions which represent possible transitions in the concrete model, and *must*-transitions [Larsen and Thomsen 1988; Dams et al. 1997] which represent definite transitions in the concrete model. May and must transitions correspond to over and under approximations, and are needed in order to preserve formulae of the form $AX\psi$ and $EX\psi$, respectively.

We consider the 3-valued semantics of CTL formulae and suggest a new game-based model checking algorithm. We would like to be able to deduce all three truth values (true, false and indefinite) from the game. To do so, we allow each player to have two roles in the new 3-valued model checking game. The goal of ∀belard is either to refute φ on M or to prevent ∃loise from verifying. Similarly, the goal of ∃loise is either to verify or to prevent ∀belard from refuting. As before, ∀belard has a winning strategy iff $M \not\models \varphi$, and ∃loise has a winning strategy iff $M \models \varphi$. However, it is also possible that neither of them has a winning strategy, in which case the value of φ in M is indefinite.

In order to check φ on the abstract model M, we propose a coloring algorithm over three colors: T, F, and ?. If all the initial nodes of the game-graph are colored by T, then we conclude that $M \models \varphi$. If some initial node of the game-graph is colored by F, we know that $M \not\models \varphi$. Both these results apply to the concrete model as well. Yet, if none of the above holds, meaning that none of the initial nodes is colored by F and at least one of them is colored by ?, we have no definite answer. It is then desirable to refine the abstract model.

We choose a criterion for refinement by examining the part of the game-graph which is colored by ?. Once a criterion for refinement is chosen, the refinement is traditionally done by splitting abstract states throughout the entire abstract model. That is, while the decision on the criterion for refinement is local, the refinement is global. In contrast to traditional refinement, the structure of the game-graph allows us to refine only the indefinite part of the model. It also allows us to use definite results that were obtained previously. Thus, model checking exploits results from

previous runs (it is incremental). In addition, the abstract model does not grow where it is not needed.

As for our second goal, we propose an algorithm that constructs an *annotated counterexample* in case model checking ends with a negative answer, meaning that the checked property φ is refuted by the examined model M. We first deal with the simpler case where model checking is applied to a concrete model. The construction uses the colored game-graph and starts from an initial node which is colored by F. If the formula in a node n is either $AX\psi$ or $\psi_1 \wedge \psi_2$ then we include in the counterexample one successor of n, which is colored by F. This successor cannot be chosen arbitrarily, but based on the coloring process. If the formula in n is either $EX\psi$ or $\psi_1 \vee \psi_2$ then we include in the counterexample all the successors of n (which are all colored by F). The resulting counterexample is an annotated sub-model of M, with possibly some unwinding, that gives the full reason for the refutation of φ on M.

Having defined the notion of an annotated counterexample, we then discuss the construction of annotated counterexamples when abstract models are used. In the 3-valued case, concretization of an abstract annotated counterexample will never fail since the 3-valued abstraction is conservative w.r.t. negative results as well. Thus, we can use an extension of the concrete algorithm to provide an abstract counterexample and derive from it a concrete one.

To conclude, the main contributions of this work are:

—A game-based CTL model checking for abstract models over the 3-valued semantics, which can be used for verification as well as refutation.

—A new notion of refinement, that eliminates indefinite results of the model checking.

—An incremental model checking within the framework of abstraction-refinement.

—A sufficient and minimal counterexample for full CTL.

1.1 Related work

Different formalisms of abstract models suitable for the 3-valued semantics are proposed in [Larsen and Thomsen 1988; Larsen 1989; Bruns and Godefroid 1999; 2000; Huth et al. 2001; Godefroid and Jagadeesan 2002]. It is shown in [Godefroid and Jagadeesan 2003] that they have the same expressiveness and, based on [Bruns and Godefroid 2000], that their model checking problem can be reduced to two traditional (2-valued) model checking problems: one for satisfaction and one for refutation. In this work we use *Kripke Modal Transition Systems* [Huth et al. 2001; Godefroid and Jagadeesan 2002], and solve the 3-valued model checking problem *directly*. The direct solution has the same complexity as traditional model checking and it becomes helpful when refinement is needed.

Using a reduction to two 2-valued problems, as suggested in [Bruns and Godefroid 2000; Godefroid and Jagadeesan 2003], results in the same answer as our algorithm. Yet, it is then not clear how to guide the refinement, in case it is needed. This is because the model used for satisfaction and the one used for refutation are not identical, thus indefinite states cannot be deduced from their intersection. Our approach for refinement uses the indefinite portion of the game-graph, hence

it is not applicable in these cases. Therefore, the application to refinement demonstrates the advantage of designing a *direct* 3-valued model checking algorithm. This subject is further discussed in Appendix A.

Other direct algorithms for 3-valued CTL model checking appeared in the literature (e.g. [Bruns and Godefroid 1999; Chechik et al. 2001; Chechik et al. 2002]). In [Bruns and Godefroid 1999], an explicit-state model checking algorithm is suggested. The abstract model is described by a *partial Kripke structure*, which allows uncertainty in the labelling of the states, but does not distinguish may and must transitions. In order to use this algorithm on a KMTS, a reduction is needed, as described in [Godefroid and Jagadeesan 2003]. In [Chechik et al. 2001], a symbolic algorithm for multi-valued CTL model checking is provided. This algorithm generalizes 3-valued CTL model checking to multi-valued CTL model checking. The algorithm is implemented by a tool called χChek [Chechik et al. 2002]. However, these works are not accompanied with a refinement mechanism. As earlier stated, it is when refinement is needed that the direct solution becomes most helpful, which stretches out the advantage of having such a mechanism.

As an enhancement of the standard 3-valued semantics, [Bruns and Godefroid 2000; Godefroid and Jagadeesan 2002] introduce the *thorough* 3-valued semantics. The thorough semantics gives more definite answers than the standard 3-valued semantics, at the expense of increasing the complexity of model checking. We use the standard 3-valued semantics [Bruns and Godefroid 1999], which is less precise, but enjoys a better complexity of the model checking algorithm, namely our algorithm has linear running time both in the size of the model and in the size of the formula.

Various abstraction techniques are formalized in the framework of abstract interpretation [Cousot and Cousot 1977; Loiseaux et al. 1995; Dams et al. 1997]. [Graf and Saidi 1997; Das et al. 1999; Saidi 2000; Namjoshi and Kurshan 2000; Saidi and Shankar 1999] use *predicate abstractions* (also called *boolean abstractions*) to construct abstract models. [Godefroid et al. 2001] discusses predicate and cartesian abstraction techniques. [Kurshan 1994; Barner et al. 2002; Clarke et al. 2002; Chauhan et al. 2002] discuss abstractions based on visible variables. [Clarke et al. 2000] define an abstraction based on variables clusters. Our work can be combined with any of these abstractions.

Localization reduction [Kurshan 1994], or iterative abstraction-refinement [Clarke et al. 2000] provides a framework for model checking of universal properties, with counterexample guided refinement. [Govindaraju and Dill 1998; Barner et al. 2002; Clarke et al. 2002; Chauhan et al. 2002] present improvements to this approach, applicable to universal properties only. We consider full CTL and do not restrict the discussion to universal properties.

Other researchers have suggested abstraction-refinement mechanisms for various branching time temporal logics. In [Lee et al. 1996] the tearing paradigm is presented as a way to obtain lower and upper approximations of the system. Yet, their technique is restricted to ACTL or ECTL. In [Pardo and Hachtel 1997; 1998] the full propositional mu-calculus is considered. In their abstraction, the concrete and abstract systems share the same state space. The simplification is based on taking supersets and subsets of a given set with a more compact BDD represen-

tation. In [Lind-Nielsen and Andersen 1999] full CTL is handled. However, the verified system has to be described as a cartesian product of machines. The initial abstraction considers only machines that directly influence the formula and in each iteration the cone of influence is extended in a BFS manner. [Asteroth et al. 2001] handles ACTL and full CTL. Their abstraction collapses all states that satisfy the same subformulae of φ into an abstract state. Thus, computing the abstract model is at least as hard as model checking. Instead, they use partial knowledge on the abstraction function and gain information in each refinement. Our approach for abstraction-refinement is designed for full CTL and is suitable for any abstraction that can be described in the framework of abstract interpretation, thus it is more general. It also has the advantage of being most suitable for using results from previous iterations, resulting in an incremental algorithm.

[Namjoshi 2001; Tan and Cleaveland 2002] investigated the idea of generating temporal proofs for branching time properties from the information gained by model checking, when the verification succeeds. Since a counterexample may be viewed as a proof of satisfaction for the *negation* of the property, these works are related to our work on counterexamples. Unlike our goal, these works are mainly aimed at finding or ruling out errors in the model checker itself. Another main difference is that we use the information gained during the run of the model checker to present a counterexample, which is an extended sub-model, rather than a deductive proof. A deductive proof may be seen as a representation of *all* the counterexamples, thus it serves different purposes.

The game-based approach to model checking, used in this work, is closely related to the Automata-theoretic approach [Kupferman et al. 2000], as described in [Leucker 1999]. Thus, our work can also be described in this framework, using alternating automata.

1.2 Organization

The rest of the paper is organized as follows. In the next section we give the necessary background for the game-based CTL model checking, abstractions and the 3-valued semantics. Due to technical reasons, we then start with the description of an annotated counterexample. Thus, in Section 3 we describe how to construct an annotated counterexample for full CTL and show that it is sufficient and minimal. In Section 4 we extend the game-based model checking algorithm to abstract models, using the 3-valued semantics. We then describe how to produce a concrete annotated counterexample using the abstract information in section 5. In Section 6 we present our refinement technique, leading to an incremental abstraction-refinement framework, which is described in Section 7. Finally, we discuss some conclusions in Section 8. Due to space limitations we defer most proofs to an electronic appendix (see also [Shoham 2003]).

2. PRELIMINARIES

Let AP be a finite set of atomic propositions. We define the set Lit of literals over AP to be the set $Lit = AP \cup \{\neg p : p \in AP\}$, i.e. for each $p \in AP$, both p and $\neg p$ are in Lit. We identify $\neg\neg p$ with p.

Definition 2.1. The Logic CTL in negation normal form is the set of formulae

defined as follows:

$$\varphi ::= \text{true} \mid \text{false} \mid l \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\psi \mid E\psi$$

where l ranges over Lit, and ψ is defined by

$$\psi ::= X\varphi \mid \varphi U \varphi \mid \varphi V \varphi$$

The (concrete) semantics of CTL formulae is defined with respect to a Kripke structure.

Definition 2.2. A *Kripke Structure* is a tuple $M = (S, S_0, \rightarrow, L)$, 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* (i.e., for every state $s \in S$ there exists a state $s' \in S$ such that $s \rightarrow s'$) and $L : S \rightarrow 2^{Lit}$ is a labeling function that associates each state in S with a subset of literals, such that for each state s and atomic proposition $p \in AP$, we have that exactly one of p and $\neg p$ is in $L(s)$, i.e. $p \in L(s)$ iff $\neg p \notin L(s)$.

A *path* in M is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \rightarrow s_{i+1}$. If $s = s_0$, then π is said to be *from* s.

$[(M, s) \models \varphi] = \text{tt}$ means that the CTL formula φ is true in the state s of a Kripke structure M. $[(M, s) \models \varphi] = \text{ff}$ means that φ is false in s. The formal definition follows.

Definition 2.3. [Clarke et al. 1999] The truth value $\in \{\text{tt}, \text{ff}\}$ of a CTL formula φ in a state s of a Kripke structure $M = (S, S_0, \rightarrow, L)$, denoted $[(M, s) \models \varphi]$, is defined inductively as follows:

$$\begin{array}{lcl}
[(M,s) \models \text{true}] & = & \text{tt} \\
[(M,s) \models \text{false}] & = & \text{ff} \\
[(M,s) \models l] & = & \text{tt} \;\Leftrightarrow\; l \in L(s), \text{ where } l \in Lit \\
[(M,s) \models \varphi_1 \wedge \varphi_2] & = & \text{tt} \;\Leftrightarrow\; [(M,s) \models \varphi_1] = \text{tt} \;\text{ and }\; [(M,s) \models \varphi_2] = \text{tt} \\
[(M,s) \models \varphi_1 \vee \varphi_2] & = & \text{tt} \;\Leftrightarrow\; [(M,s) \models \varphi_1] = \text{tt} \;\text{ or }\; [(M,s) \models \varphi_2] = \text{tt} \\
[(M,s) \models A\psi] & = & \text{tt} \;\Leftrightarrow\; \forall \pi \text{ from } s : [(M,\pi) \models \psi] = \text{tt} \\
[(M,s) \models E\psi] & = & \text{tt} \;\Leftrightarrow\; \exists \pi \text{ from } s : [(M,\pi) \models \psi] = \text{tt}
\end{array}$$

For a path $\pi = s_0, s_1, \ldots$, $[(M, \pi) \models \psi]$ is defined as follows.

$$\begin{array}{lcl}
[(M,\pi) \models X\varphi] & = & [(M, s_1) \models \varphi] \\
[(M,\pi) \models \varphi_1 U \varphi_2] & = & \text{tt} \;\Leftrightarrow\; \exists k \geq 0 : [([(M, s_k) \models \varphi_2] = \text{tt}) \\
 & & \qquad \wedge (\forall j < k : [(M, s_j) \models \varphi_1] = \text{tt})] \\
[(M,\pi) \models \varphi_1 V \varphi_2] & = & \text{tt} \;\Leftrightarrow\; \forall k \geq 0 : [(\forall j < k : [(M, s_j) \models \varphi_1] = \text{ff}) \\
 & & \qquad \Rightarrow ([(M, s_k) \models \varphi_2] = \text{tt})]
\end{array}$$

We say that M satisfies φ, denoted $[M \models \varphi] = \text{tt}$, if $\forall s_0 \in S_0 : [(M, s_0) \models \varphi] = \text{tt}$. Otherwise, M refutes φ, denoted $[M \models \varphi] = \text{ff}$.

$\varphi_1 U \varphi_2$ can be read as "φ_1 *until* φ_2", where φ_2 must hold at some point along the path. Its dual, $\varphi_1 V \varphi_2$, can be read as "φ_2 *while not* φ_1", where it is possible that φ_1 never holds. When M is clear from the context, we omit it from the notation and write $[s \models \varphi]$ or $[\pi \models \psi]$.

Definition 2.4. Given a CTL formula φ of the form $A(\varphi_1 U \varphi_2)$, $E(\varphi_1 U \varphi_2)$, $A(\varphi_1 V \varphi_2)$ or $E(\varphi_1 V \varphi_2)$, its *expansion* $exp(\varphi)$ is defined as:

$$\begin{array}{lcl}
\varphi = A(\varphi_1 U \varphi_2) & : & exp(\varphi) = \{\varphi,\ \varphi_2 \vee (\varphi_1 \wedge AX\varphi),\ \varphi_1 \wedge AX\varphi,\ AX\varphi\} \\
\varphi = E(\varphi_1 U \varphi_2) & : & exp(\varphi) = \{\varphi,\ \varphi_2 \vee (\varphi_1 \wedge EX\varphi),\ \varphi_1 \wedge EX\varphi,\ EX\varphi\} \\
\varphi = A(\varphi_1 V \varphi_2) & : & exp(\varphi) = \{\varphi,\ \varphi_2 \wedge (\varphi_1 \vee AX\varphi),\ \varphi_1 \vee AX\varphi,\ AX\varphi\} \\
\varphi = E(\varphi_1 V \varphi_2) & : & exp(\varphi) = \{\varphi,\ \varphi_2 \wedge (\varphi_1 \vee EX\varphi),\ \varphi_1 \vee EX\varphi,\ EX\varphi\}
\end{array}$$

2.1 Game-based Model Checking Algorithm

In this section we present the Game-theoretic approach to Model Checking of CTL formulae in a (concrete) Kripke structure [Stirling 2001; Leucker 1999]. Given a Kripke structure $M = (S, S_0, \rightarrow, L)$ and a CTL formula φ, the *model checking game* of M and φ is defined as follows. Its board is the Cartesian product $S \times sub(\varphi)$ of the set of states S and the set of subformulae $sub(\varphi)$, where $sub(\varphi)$ is defined by:
if $\varphi =$ true, false or l where $l \in Lit$ then $sub(\varphi) = \{\varphi\}$.
if $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ then $sub(\varphi) = \{\varphi\} \cup sub(\varphi_1) \cup sub(\varphi_2)$.
if $\varphi = AX\varphi_1$ or $EX\varphi_1$ then $sub(\varphi) = \{\varphi\} \cup sub(\varphi_1)$.
if $\varphi = A(\varphi_1 U \varphi_2)$, $E(\varphi_1 U \varphi_2)$, $A(\varphi_1 V \varphi_2)$ or $E(\varphi_1 V \varphi_2)$ then $sub(\varphi) = exp(\varphi) \cup sub(\varphi_1) \cup sub(\varphi_2)$.

Given a state $s \in S$, the model checking game is played by two players, $\forall$belard, the refuter who wants to show that $[(M, s) \models \varphi] =$ ff, and $\exists$loise, the prover who wants to show that $[(M, s) \models \varphi] =$ tt. A single *play* from (s, φ) is a (possibly infinite) sequence $C_0 \rightarrow_{p0} C_1 \rightarrow_{p1} C_2 \rightarrow_{p2} ...$ of configurations, where $C_0 = (s, \varphi)$, $C_i \in S \times sub(\varphi)$ and $p_i \in \{\forall\text{belard}, \exists\text{loise}\}$. The subformula in C_i determines which player p_i makes the next move.

The possible moves at each step are:

(1) $C_i = (s, \text{false})$, $C_i = (s, \text{true})$, or $C_i = (s, l)$ where $l \in Lit$: the play is finished. Such configurations are called *terminal configurations*.
(2) $C_i = (s, AX\varphi)$: $\forall$belard chooses a transition $s \rightarrow s'$ in M and $C_{i+1} = (s', \varphi)$.
(3) $C_i = (s, EX\varphi)$: $\exists$loise chooses a transition $s \rightarrow s'$ in M and $C_{i+1} = (s', \varphi)$.
(4) $C_i = (s, \varphi_1 \wedge \varphi_2)$: $\forall$belard chooses $j \in \{1, 2\}$ and $C_{i+1} = (s, \varphi_j)$.
(5) $C_i = (s, \varphi_1 \vee \varphi_2)$: $\exists$loise chooses $j \in \{1, 2\}$ and $C_{i+1} = (s, \varphi_j)$.
(6) $C_i = (s, A(\varphi_1 U \varphi_2))$: $C_{i+1} = (s, \varphi_2 \vee (\varphi_1 \wedge AXA(\varphi_1 U \varphi_2)))$.
(7) $C_i = (s, E(\varphi_1 U \varphi_2))$: $C_{i+1} = (s, \varphi_2 \vee (\varphi_1 \wedge EXE(\varphi_1 U \varphi_2)))$.
(8) $C_i = (s, A(\varphi_1 V \varphi_2))$: $C_{i+1} = (s, \varphi_2 \wedge (\varphi_1 \vee AXA(\varphi_1 V \varphi_2)))$.
(9) $C_i = (s, E(\varphi_1 V \varphi_2))$: $C_{i+1} = (s, \varphi_2 \wedge (\varphi_1 \vee EXE(\varphi_1 V \varphi_2)))$.

In configurations 6-9 the move is deterministic, thus any player can make the move.

A play is *maximal* iff it is infinite or ends in a terminal configuration. In [Stirling 2001] it has been shown that a play is infinite iff there is exactly one subformula of the form AU, EU, AV or EV that occurs infinitely often in the play. Such a subformula is called a *witness*.

Winning Criteria: $\forall$belard wins a (maximal) play iff one of the following holds:

(1) the play is finite and ends in a terminal configuration of the form $C_i = (s, \text{false})$, or $C_i = (s, l)$, where $l \notin L(s)$.
(2) the play is infinite and the witness is of the form AU or EU.

∃loise wins the (maximal) play otherwise, i.e. iff one of the following holds:

(1) the play is finite and ends in a terminal configuration of the form $C_i = (s, \text{true})$, or $C_i = (s, l)$, where $l \in L(s)$.
(2) the play is infinite and the witness is of the form AV or EV.

The model checking *game* from (s, φ) consists of all the possible plays from (s, φ). A *strategy* is a set of rules for a player, telling the player which move to choose in the current configuration. A *winning strategy* from (s, φ) is a set of rules allowing the player to win every play starting at (s, φ) if he plays by the rules. The following theorem tells us that the model checking problem can be reduced to the problem of finding which player has a winning strategy in the model checking game.

THEOREM 2.5. *[Stirling 2001] Let M be a Kripke structure and φ a CTL formula. Then, for each $s \in S$:*

(1) $[(M, s) \models \varphi] = tt$ *iff* *∃loise has a winning strategy starting at* (s, φ).
(2) $[(M, s) \models \varphi] = ff$ *iff* *∀belard has a winning strategy starting at* (s, φ).

The model checking algorithm for the evaluation of $[M \models \varphi]$ consists of two parts. First, it constructs (part of) the *game-graph*. The *game-graph* is the graph whose nodes are the elements (configurations) of the game board and whose edges are the possible moves of the players. It captures all the possible plays of a game (from any configuration). The evaluation of the truth value of φ in M is then done in the second phase of the algorithm by coloring the game-graph. This phase is described in Section 2.1.2.

2.1.1 *Game-Graph Construction and its Properties.* The truth value of φ in M depends on its truth value in the initial states of M. Thus, we are interested in plays that start from configurations in $S_0 \times \{\varphi\}$, referred to as *initial configurations*. The subgraph of the game-graph that is reachable from the *initial configurations* $S_0 \times \{\varphi\}$ is constructed in a BFS or DFS manner. The construction starts from the initial configurations (nodes) and applies each possible move, by the previously described rules, to get the successors in the game-graph of each new node. The result is denoted $G_{M \times \varphi} = (N, E)$, where $N \subseteq S \times sub(\varphi)$ is the set of nodes of $G_{M \times \varphi}$, and $E \subseteq N \times N$ is the set of edges. N contains a node for each configuration that was reached during the construction and E contains an edge for each possible move that was applied.

The nodes (configurations) of the game-graph can be classified into three types.

(1) Terminal configurations are leaves in the game-graph.
(2) Nodes whose formula is of the form $\varphi_1 \wedge \varphi_2$ or $AX\varphi_1$ are *∧-nodes*.
(3) Nodes whose formula is of the form $\varphi_1 \vee \varphi_2$ or $EX\varphi_1$ are *∨-nodes*.

Nodes whose formula is of the form AU, EU, AV, EV can be considered either ∨-nodes or ∧-nodes. This is because such nodes have only one son, and the treatment of ∧-nodes and ∨-nodes in the case of one son is always the same. Sometimes we further distinguish between nodes whose formula is of the form $AX\varphi$ ($EX\varphi$) and other ∧-nodes (∨-nodes) by referring to them as *AX-nodes* (*EX-nodes*). The edges in the game-graph are also divided into two types.

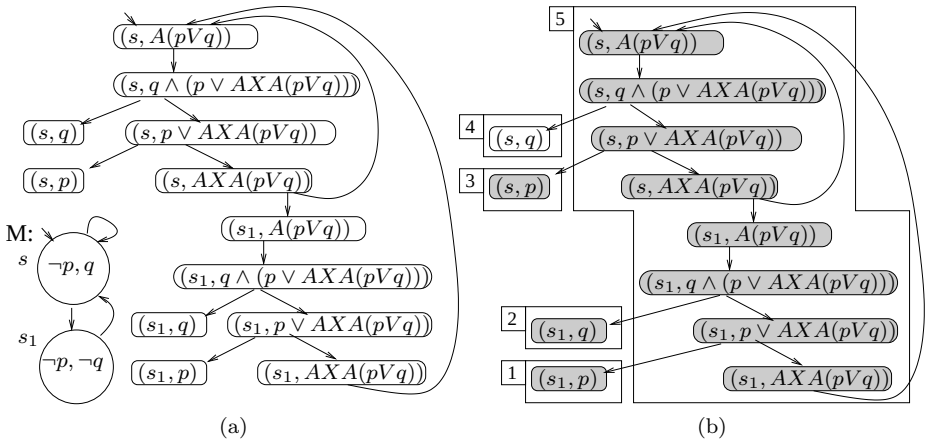

Fig. 1. (a) A game-graph for M and $\varphi = A(pVq)$, and (b) Its coloring, where rectangles depict the partition of the nodes, white nodes are colored by T and grey nodes are colored by F.

—Edges that originate in AX-nodes or EX-nodes are *progress* edges that reflect real transitions of the Kripke structure.

—Other edges are *auxiliary* edges.

An important property of the game-graph is described by the following lemma.

LEMMA 2.6. *Let B be a non-trivial strongly connected component (SCC)*[2] *in a game-graph. Then the set of formulae that are associated with the nodes in B is exactly one of the sets* $\exp(\varphi)$*, where* $\varphi \in \{A(\varphi_1 U \varphi_2), E(\varphi_1 U \varphi_2), A(\varphi_1 V \varphi_2), E(\varphi_1 V \varphi_2)\}$.

Based on Lemma 2.6, we generalize the notion of a witness in the context of the game-graph. The formula φ such that $exp(\varphi)$ is the set of formulae in a non-trivial SCC is called a *witness*. Each non-trivial SCC is classified as an AU, AV, EU, or EV SCC, based on its witness.

Example 2.7. Figure 1(a) shows an example of a game-graph G, constructed for the formula $\varphi = A(pVq)$ and the given model M. The model M has a single initial state s, thus G has a single initial node $(s, A(pVq))$, pointed by a small arrow. This example also demonstrates Lemma 2.6. G has two non-trivial SCCs. The set of formulae in each one of them is exactly $\exp(A(pVq))$, making them classified as AV-SCCs.

2.1.2 *Coloring Algorithm.* The following *Coloring Algorithm* [Bollig et al. 2002] labels each node (configuration) in the game-graph $G_{M\times\varphi}$ by T or F, depending on whether ∃loise or ∀belard has a winning strategy for the game that starts at that node.

The game-graph is partitioned into its *Maximal Strongly Connected Components* (MSCCs), denoted Q_i's, and an order $\leq$ is determined on the Q_i's, such that an

[2]A non-trivial SCC contains at least one edge. Unless otherwise stated, it is not necessarily maximal.

edge (n, n'), where $n \in Q_i$ and $n' \in Q_j$, exists in the game-graph only if $Q_j \leq Q_i$. Such an order exists because the MSCCs of the game-graph form a *directed acyclic graph* (DAG). It is extended to a total order $\leq$ arbitrarily.

The coloring algorithm processes the Q_i's according to the determined order, bottom-up. Let Q_i be the smallest MSCC with respect to $\leq$ that is not yet fully colored. Hence, every outgoing edge of a node in Q_i leads either to a colored node or to a node in the same set, Q_i. The nodes of Q_i are colored as follows.

(1) Terminal nodes in Q_i are colored by T if $\exists$loise wins in them, and by F otherwise.
(2) An $\vee$-node is colored by T if it has a son that is colored T, and by F if all its sons are colored F.
(3) An $\wedge$-node is colored by T if all its sons are colored T, and by F if it has a son that is colored F.
(4) All the nodes in Q_i that remain uncolored after the propagation of these rules are colored according to the witness in Q_i (by Lemma 2.6 there exists exactly one such witness). They are colored by F if the witness is of the form AU or EU, and are colored by T if the witness is of the form AV or EV.

The result of the coloring algorithm is a *coloring function* $\chi : N \to \{T, F\}$.

THEOREM 2.8. *[Stirling 2001] Let $G_{M\times\varphi}$ be a game-graph, colored by the above coloring algorithm with χ being the coloring function, and let n be a node in the game-graph, then:*

(1) $\chi(n) = T$ *iff* $\exists$*loise has a winning strategy for every play starting at* n.
(2) $\chi(n) = F$ *iff* $\forall$*belard has a winning strategy for every play starting at* n.

As a conclusion of Theorem 2.5 and Theorem 2.8, we get the following theorem.

THEOREM 2.9. *[Stirling 2001] Let M be a Kripke structure and φ a CTL formula. Then, for each $n = (s, \varphi_1) \in G_{M\times\varphi}$:*

(1) $[(M, s) \models \varphi_1] = tt$ *iff* $n = (s, \varphi_1)$ *is colored by* T.
(2) $[(M, s) \models \varphi_1] = ff$ *iff* $n = (s, \varphi_1)$ *is colored by* F.

Based on Theorem 2.9 we conclude that if every initial configuration $n_0 \in S_0 \times \{\varphi\}$ is colored by T, then $[M \models \varphi] = \text{tt}$. Otherwise, $[M \models \varphi] = \text{ff}$.

Example 2.10. Figure 1(b) provides a coloring example for the game-graph of Example 2.7. The partition to Q_i's is depicted by rectangles and the order among them can be seen by their serial numbers. The nodes in Q_1-Q_4 are colored as terminal nodes (rule 1). The nodes in Q_5 are all colored by rules 2 and 3, starting from $(s_1, q \wedge (p \vee AXA(pVq)))$ that is colored F based on its son (s_1, q). This causes $(s_1, A(pVq))$ to be colored F and the F-color continues to propagate "up" (from son to parent) until all the nodes of Q_5 are colored. The final coloring function can be seen in the figure, where white nodes are colored by T and grey nodes are colored by F. The initial node $(s, A(pVq))$ is colored F, which means that the given model M refutes $A(pVq)$.

2.2 Abstraction

In this section we present abstract models (for CTL) and their relation to concrete models. So far, we considered Kripke structures that represent concrete models, and discussed the semantics of CTL formulae with respect to them. However, concrete Kripke structures may be very large. Consequently, their model checking problem becomes infeasible due to the state explosion problem. A powerful solution is based on using abstractions of the concrete model.

It turns out that in order to guarantee preservation of CTL formulae from abstract models to concrete models, we need to introduce *two* transition relations [Larsen and Thomsen 1988; Dams et al. 1997]: preservation of *universal* properties requires an over-approximation, whereas preservation of *existential* properties requires an under-approximation. This is accomplished by using *Kripke Modal Transition Systems* [Huth et al. 2001; Godefroid and Jagadeesan 2002].

Definition 2.11. A *Kripke Modal Transition System* (KMTS) is a tuple $M = (S, S_0, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\xrightarrow{\text{must}} \subseteq S \times S$ and $\xrightarrow{\text{may}} \subseteq S \times S$ are transition relations such that the relation $\xrightarrow{\text{may}}$ is *total* and $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$, and $L : S \to 2^{Lit}$ is a labeling function that associates each state in S with literals from Lit, such that for each state s and atomic proposition $p \in AP$, *at most* one of p and $\neg p$ is in $L(s)$.

A *must (may)* path in M is a *maximal* sequence of states, $\pi = s_0, s_1, \ldots$ such that for every two consecutive states s_i, s_{i+1} in π, 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 π cannot be extended by any other transition of the same type. If $s = s_0$, then π is said to be *from* s.

Transitions in $\xrightarrow{\text{must}}$ are called *must* transitions, and transitions in $\xrightarrow{\text{may}}$ are called *may* transitions. Note that since the transition relation $\xrightarrow{\text{may}}$ is total, then every may path is infinite (due to the maximality), whereas a (maximal) must path can be finite, since the transition relation $\xrightarrow{\text{must}}$ is not necessarily total. This means that although every must-transition is also a may-transition (by definition), the same does *not* necessarily hold for paths. That is, a finite must path is not considered a may path, because it is not maximal in terms of may transitions. Since we now consider finite paths in addition to infinite paths, we need the following definition.

Definition 2.12. Let π be a must or may path. The *length* of π, denoted $|\pi|$, is defined to be the number of states in π. That is,

$$|\pi| = \begin{cases} \infty & \text{if } \pi \text{ is infinite} \\ n+1 & \text{if } \pi \text{ is finite and of the form } s_0, \ldots, s_n \end{cases}$$

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 atomic proposition $p \in AP$, we have that exactly one of p and $\neg p$ is in $L(s)$.

We consider abstractions that are done by collapsing sets of concrete states (from S_C) into single abstract states (in S_A). Such abstractions can be described in the framework of *Abstract Interpretation* [Loiseaux et al. 1995; Dams et al. 1997].

Definition 2.13. [Cousot and Cousot 1977; Loiseaux et al. 1995] Let $(C, \preceq)$ and $(A, \sqsubseteq)$ be two partially ordered sets (posets). $(\alpha : C \to A, \gamma : A \to C)$ is a *Galois*

connection from $(C, \preceq)$ to $(A, \sqsubseteq)$ iff (1) α and γ are total and monotonic, (2) for all $c \in C$, $\gamma \circ \alpha(c) \succeq c$, and (3) for all $a \in A$, $\alpha \circ \gamma(a) \sqsubseteq a$.

Let $M_C = (S_C, S_{0C}, \rightarrow, L_C)$ be a (concrete) Kripke structure. Let $(S_A, \sqsubseteq)$ be a poset of *abstract states* and $(\gamma : S_A \rightarrow 2^{S_C}, \alpha : 2^{S_C} \rightarrow S_A)$ a *Galois connection* from $(2^{S_C}, \subseteq)$ to $(S_A, \sqsubseteq)$, that determines its relation to the concrete states. γ is the *concretization function* that maps each abstract state to the set of concrete states that it represents. α is the *abstraction function* that maps each set of concrete states to the abstract state that represents it.

An abstract model $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$ can then be defined as follows. The set of initial abstract states S_{0A} is built such that each concrete initial state is represented by an abstract initial state and there are no additional initial abstract states, i.e. $s_{0a} \in S_{0A}$ *iff* there exists $s_{0c} \in S_{0C}$ such that $s_{0c} \in \gamma(s_{0a})$. The requirement that there are no additional initial abstract states is needed to ensure preservation of falsity in the *model*, as described in the second part of Theorem 2.16. This requirement is not needed for state-wise preservation, described in the first part of the theorem.

The labeling of an abstract state is done according to the labeling of all the concrete states that it represents. An abstract state s_a is labeled by $l \in Lit$, only if all the concrete states that are represented by it are labeled by l as well. Therefore, it is possible that neither p nor $\neg p$ are in $L_A(s_a)$.

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 origin abstract state: a must-transition $s_a \xrightarrow{\text{must}} s'_a$ exists only if $\forall s_c \in \gamma(s_a)$ we have that $\exists s'_c \in \gamma(s'_a)$ 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 *accurate*, as long as they maintain these conditions. Also note, that since the concrete transition relation is total, then the resulting abstract transition relation $\xrightarrow{\text{may}}$ is also total, as required. The abstract transition relation $\xrightarrow{\text{must}}$, on the other hand, is not necessarily total. In fact, it can even be empty.

Other constructions of abstract models, based on Galois connections, can be found in [Dams et al. 1997; Godefroid et al. 2001].

The resulting abstract model is *more abstract* than M_C as defined by the following definition, which formalizes the relation between an abstract model and a concrete model that guarantees preservation of CTL formulae.

Definition 2.14. [Larsen and Thomsen 1988; Dams et al. 1997; Godefroid and Jagadeesan 2002] Let $M_C = (S_C, S_{0C}, \rightarrow, L_C)$ be a concrete Kripke structure, and let $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$, be an abstract KMTS. We say that $H \subseteq S_C \times S_A$ is a *mixed simulation* from M_C to M_A if $(s_c, s_a) \in H$ implies the following:

(1) $L_A(s_a) \subseteq L_C(s_c)$.
(2) if $s_c \rightarrow s'_c$, then there is some $s'_a \in S_A$ such that $s_a \xrightarrow{\text{may}} s'_a$ and $(s'_c, s'_a) \in H$.
(3) if $s_a \xrightarrow{\text{must}} s'_a$, then there is some $s'_c \in S_C$ such that $s_c \rightarrow s'_c$ and $(s'_c, s'_a) \in H$.

If there exists a mixed simulation H such that for each $s_c \in S_{0C}$ there exists $s_a \in S_{0A}$ such that $(s_c, s_a) \in H$ and for each $s_a \in S_{0A}$ there exists $s_c \in S_{0C}$ such that $(s_c, s_a) \in H$, we say that M_A is *more abstract* than M_C, denoted $M_C \preceq M_A$.

The mixed simulation relation $H \subseteq S_C \times S_A$ from M_C to an abstract model which is constructed based on a Galois connection as described above is induced by the concretization function as follows. H is defined such that $(s_c, s_a) \in H$ iff $s_c \in \gamma(s_a)$. The results presented in this paper are applicable to *any* abstract model that is *more abstract* than the concrete model M_C with respect to the mixed simulation relation, and are not limited to our construction of an abstract model.

[Huth et al. 2001] defines the *3-valued semantics* of a CTL formula over a KMTS. The 3-valued semantics is designed to be *conservative* in the sense that it preserves both satisfaction (tt) and refutation (ff) of a formula from the abstract model to the concrete one. However, a new truth value, $\perp$ is introduced. If the truth value of a formula in an abstract model is $\perp$, the meaning is that its value over the concrete model it represents is not known and can be either tt or ff.

Definition 2.15. The 3-valued semantics of a CTL formula φ in a state s of a KMTS $M = (S, S_0, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L)$, denoted $[(M, s) \models^3 \varphi]$, is defined inductively as follows:

$$
\begin{array}{lcl}
[(M,s) \models^3 \text{true}] & = & \text{tt} \\
[(M,s) \models^3 \text{false}] & = & \text{ff} \\
[(M,s) \models^3 l] & = & \begin{cases} \text{tt} & \text{if } l \in L(s) \\ \text{ff} & \text{if } \neg l \in L(s) \\ \perp & \text{otherwise} \end{cases} \\
[(M,s) \models^3 \varphi_1 \wedge \varphi_2] & = & \begin{cases} \text{tt} & \text{if } [(M,s) \models^3 \varphi_1] = \text{tt and } [(M,s) \models^3 \varphi_2] = \text{tt} \\ \text{ff} & \text{if } [(M,s) \models^3 \varphi_1] = \text{ff or } [(M,s) \models^3 \varphi_2] = \text{ff} \\ \perp & \text{otherwise} \end{cases} \\
[(M,s) \models^3 \varphi_1 \vee \varphi_2] & = & \begin{cases} \text{tt} & \text{if } [(M,s) \models^3 \varphi_1] = \text{tt or } [(M,s) \models^3 \varphi_2] = \text{tt} \\ \text{ff} & \text{if } [(M,s) \models^3 \varphi_1] = \text{ff and } [(M,s) \models^3 \varphi_2] = \text{ff} \\ \perp & \text{otherwise} \end{cases} \\
[(M,s) \models^3 A\psi] & = & \begin{cases} \text{tt} & \text{if for each may-path } \pi \text{ from } s : [(M,\pi) \models^3 \psi] = \text{tt} \\ \text{ff} & \text{if there exists a must-path } \pi \text{ from } s \text{ such that :} \\ & \qquad [(M,\pi) \models^3 \psi] = \text{ff} \\ \perp & \text{otherwise} \end{cases} \\
[(M,s) \models^3 E\psi] & = & \begin{cases} \text{tt} & \text{if there exists a must-path } \pi \text{ from } s \text{ such that :} \\ & \qquad [(M,\pi) \models^3 \psi] = \text{tt} \\ \text{ff} & \text{if for each may-path } \pi \text{ from } s : [(M,\pi) \models^3 \psi] = \text{ff} \\ \perp & \text{otherwise} \end{cases}
\end{array}
$$

For a may or must path $\pi = s_0, s_1, \ldots$, $[(M, \pi) \models^3 \psi]$ is defined as follows.

$$[(M,\pi) \models^3 X\varphi] = \begin{cases} [(M,s_1) \models^3 \varphi] & \text{if } |\pi| > 1 \\ \bot & \text{otherwise} \end{cases}$$

$$[(M,\pi) \models^3 \varphi_1 U \varphi_2] = \begin{cases} \mathrm{tt} \text{ if } \exists\, 0 \le k < |\pi| : ([[(M,s_k) \models^3 \varphi_2] = \mathrm{tt}) \\ \qquad\qquad \wedge (\forall j < k : [(M,s_j) \models^3 \varphi_1] = \mathrm{tt})] \\ \mathrm{ff} \text{ if } \big(\forall\, 0 \le k < |\pi| : [(\forall j < k : [(M,s_j) \models^3 \varphi_1] \neq \mathrm{ff}) \\ \qquad\qquad \Rightarrow ([(M,s_k) \models^3 \varphi_2] = \mathrm{ff})]\big) \\ \quad \wedge \big((\forall\, 0 \le k < |\pi| : [(M,s_k) \models^3 \varphi_1] \neq \mathrm{ff}) \Rightarrow |\pi| = \infty\big) \\ \bot \text{ otherwise} \end{cases}$$

$$[(M,\pi) \models^3 \varphi_1 V \varphi_2] = \begin{cases} \mathrm{tt} \text{ if } \big(\forall\, 0 \le k < |\pi| : [(\forall j < k : [(M,s_j) \models^3 \varphi_1] \neq \mathrm{tt}) \\ \qquad\qquad \Rightarrow ([(M,s_k) \models^3 \varphi_2] = \mathrm{tt})]\big) \\ \quad \wedge \big((\forall\, 0 \le k < |\pi| : [(M,s_k) \models^3 \varphi_1] \neq \mathrm{tt}) \Rightarrow |\pi| = \infty\big) \\ \mathrm{ff} \text{ if } \exists\, 0 \le k < |\pi| : ([[(M,s_k) \models^3 \varphi_2] = \mathrm{ff}) \\ \qquad\qquad \wedge (\forall j < k : [(M,s_j) \models^3 \varphi_1] = \mathrm{ff})] \\ \bot \text{ otherwise} \end{cases}$$

We say that $[M \models^3 \varphi] = \mathrm{tt}$ if $\forall s_0 \in S_0 : [(M,s_0) \models^3 \varphi] = \mathrm{tt}$. We say that $[M \models^3 \varphi] = \mathrm{ff}$ if $\exists s_0 \in S_0 : [(M,s_0) \models^3 \varphi] = \mathrm{ff}$. Otherwise, $[M \models^3 \varphi] = \bot$.

Intuitively, the 3-valued semantics is defined such that a formula is evaluated to tt or ff only when the abstract information suffices to determine such a definite truth value that will hold in the represented concrete model. Therefore, truth of universal formulae (of the form $A\psi$) is examined along *all* the may paths (which represent at least all the concrete paths), whereas falsity of such formulae is shown by a *single* must path (which represents a definite concrete path), and dually for existential formulae (of the form $E\psi$). Similar arguments apply to the evaluation of path formulae of the form $X\varphi,\ \varphi_1 U \varphi_2,\ \varphi_1 V \varphi_2$. For example, in order to say that the truth value of $\varphi_1 U \varphi_2$ in a path is true, φ_2 needs to become true *within* the path. In order to say that $\varphi_1 U \varphi_2$ is false in the path we require that at every position, if φ_1 is not false yet, then φ_2 is still false (the eventuality is not fulfilled), and we also require that if φ_1 is *never* false, then the path is infinite. The latter requirement is needed because if the situation where φ_1 is never false happens in a *finite* path, then in the concrete model that is represented by the abstract KMTS there is still "hope" that φ_2 will become true in the future, making the until formula true. The reason is that real concrete paths are infinite. This leaves us with two possibilities for falsity of $\varphi_1 U \varphi_2$: either φ_1 becomes false *within* the path (when φ_2 is still false as well), or the path is infinite and φ_2 is false all along. This definition resembles the semantics presented in [Eisner et al. 2003] with respect to (concrete) truncated paths under the *strong* context.

The preservation of CTL formulae from an abstract model to a concrete model is guaranteed by the following theorem.

THEOREM 2.16. *[Godefroid and Jagadeesan 2002] Let $H \subseteq S_C \times S_A$ be a mixed simulation relation from a Kripke structure M_C to a KMTS M_A. Then for every $(s_c, s_a) \in H$ and every CTL formula φ, we have that:*

(1) $[(M_A, s_a) \models^3 \varphi] = tt$ *implies that* $[(M_C, s_c) \models \varphi] = tt$.

(2) $[(M_A, s_a) \overset{3}{\models} \varphi] = ff$ *implies that* $[(M_C, s_c) \models \varphi] = ff$.

We conclude that if $M_C \preceq M_A$*, then for every CTL formula* φ*, we have that:*

(1) $[M_A \overset{3}{\models} \varphi] = tt$ *implies that* $[M_C \models \varphi] = tt$.

(2) $[M_A \overset{3}{\models} \varphi] = ff$ *implies that* $[M_C \models \varphi] = ff$.

3. USING GAMES TO PRODUCE ANNOTATED COUNTEREXAMPLES

In this section we consider the concrete semantics of CTL. Our goal here is to define and produce CTL counterexamples. The usual notion of a counterexample as a path works for linear-time properties, but not for branching-time properties. Having to explain falsity of both existential and universal properties, a CTL counterexample should really be thought of as a proof of the negation of the property of interest, presented over the model at hand. We suggest the use of an *annotated counterexample* to achieve this goal.

We first informally describe our requirements of a counterexample. Traditionally, a counterexample is a sub-model, possibly with some unwinding. It is expected to:

(1) falsify the given formula.
(2) hold "enough" information to explain why the original model does not satisfy the formula.
(3) be minimal, in the sense that every state and transition are needed to maintain 1 and 2.

The minimality that is expected of the counterexample is in the sense that we wish to have *precise* counterexamples, without redundancies. For example, if a finite prefix of a path suffices to prove the refutation, we would like to see only this prefix rather than the entire (infinite) path. Note that we do not refer to shortest (or smallest) counterexamples and this should not be confused with minimality.

For CTL, having a sub-model as a counterexample is probably not what we want. We need it to be annotated with further explanatory information. Such information naturally appears in a game-graph, that was constructed based on the checked property and the relevant model: in the game-graph each node is marked by a state s and by a subformula φ_1, with the meaning that the truth value of φ_1 is examined in state s. The outgoing edges point out the nodes on which this truth value depends.

Therefore, we define a CTL annotated counterexample to be a subgraph of the corresponding game-graph, rather than a sub-model. The above requirements of a counterexample need to be adapted accordingly. Let χ denote the coloring function of the game-graph. Rephrasing the requirements of a counterexample in terms of a subgraph leads to the following expectations of an annotated counterexample, which correspond to the above three.

(1) It contains an initial node n_0 which is colored F by χ.
(2) It holds "enough" information to explain why n_0 is colored by F.
(3) It is minimal, in the sense that every node and edge are needed to maintain the previous requirements.

In order to formalize the second requirement with respect to an annotated counterexample, we need the following definitions.

Definition 3.1. Let $G = (N, E)$ be a game-graph and let A be a subgraph of G. The *partial coloring algorithm of G with respect to A* works as follows. It is given an *initial coloring function* $\chi_I : N \setminus A \to \{T, F\}$ and computes a coloring function for G. The algorithm is identical to the (original) coloring algorithm, except for the addition of the following rule:

—A node $n \in N \setminus A$ is colored by $\chi_I(n)$ and its color is not changed as a result of other rules.

Any result of the partial coloring algorithm of G with respect to A is called a *partial coloring function of G with respect to A*, denoted $\overline{\chi} : N \to \{T, F\}$.

As opposed to the usual coloring algorithm that has only one possible result, referred to as the coloring function of the game-graph, the partial coloring algorithm has several possible results, depending on the initial coloring function χ_I. Each one of them is considered a partial coloring function of the game-graph w.r.t. A. By definition, the usual coloring algorithm is a partial coloring algorithm of G with respect to G.

Definition 3.2. Let G be a game-graph and let χ be the result of the coloring algorithm on G. A subgraph A of G is *independent of* G if for each $\overline{\chi}$ that is a partial coloring function of G with respect to A, and for each $n \in A$, we have that $\chi(n) = \overline{\chi}(n)$.

Basically, a subgraph is independent of a game-graph if its coloring is *absolute* in the sense that every completion of its coloring to the full game-graph does not change the color of any node in the subgraph. In fact, one may notice that the colors of terminal nodes determine the coloring function of the full game-graph. Thus, to capture the notion of an independent subgraph, it suffices to refer to a partial coloring algorithm that allows arbitrary coloring of the terminal nodes in $N \setminus A$, but maintains the consistency of the coloring of the rest of the nodes. However, for simplicity, we strengthen the definition and allow non-deterministic coloring of all the nodes in $N \setminus A$.

The notion of an independent subgraph captures our second expectation of an annotated counterexample, since the coloring of such a subgraph determines in a definite manner the coloring of any node within it such that other parts of the game-graph can not affect or change it. This applies in particular to the initial node n_0, that by the first expectation of an annotated counterexample is included in the subgraph. Thus, such a subgraph holds sufficient information for explaining the color of the initial node n_0.

To formalize the third requirement, we use the following definition of minimality.

Definition 3.3. Let A be a subgraph of some graph, and let $P_1, \ldots, P_n$ be properties of A. We say that A is *minimal* w.r.t. $P_1, \ldots, P_n$ if there exists no strict subgraph A' of A that fulfills $P_1, \ldots, P_2$.

Having formalized the second and third requirements, we can now formalize the notion of an annotated counterexample. Yet, before doing so, we note that since

we are dealing with formulae in negation normal form, the properties of the game-graph imply that a subgraph that fulfills the three requirements described above is in fact *entirely* colored by F (rather than just having an initial node n_0 that is colored by F). This is expressed by the following lemma.

LEMMA 3.4. *Let G be a game-graph, χ its coloring function and A a subgraph of G with the following properties: (1) It contains an initial node n_0, colored F by χ, (2) It is independent of G, and (3) It is minimal w.r.t. 1 and 2. Then, for* every *$n \in A$: $\chi(n) = F$.*

We use the observation described in Lemma 3.4 and define an annotated counterexample as follows.

Definition 3.5. Let G be a game-graph, and let χ be its coloring function, such that $\chi(n_0) = F$ for some initial node n_0. A subgraph $\tilde{C}$ of G containing n_0 is an *annotated counterexample* if it satisfies the following conditions.

(1) For each node $n \in \tilde{C}$, $\chi(n) = F$.
(2) $\tilde{C}$ is independent of G.
(3) $\tilde{C}$ is minimal w.r.t. 1 and 2.

The first two requirements in Definition 3.5 imply that $\tilde{C}$ is *sufficient* for explaining why the initial node is colored by F. First it guarantees that all the nodes in $\tilde{C}$ are colored by F. In addition, since $\tilde{C}$ is independent of G, we can conclude that regardless of the other nodes in G, all the nodes in $\tilde{C}$, and in particular the initial node n_0, will be colored by F. Therefore, it also explains why the formula is refuted by the model. The third condition shows that $\tilde{C}$ is also "necessary".

Note that an annotated counterexample can be viewed as a (minimal) part of the winning strategy of the refuter that is sufficient to guarantee its victory.

3.1 Constructing an Annotated Counterexample

In this section we describe how to construct an *annotated counterexample* from a game-graph for a Kripke structure M and a CTL formula φ, as well as the information gained by the coloring algorithm, in case M does not satisfy φ.

First, the coloring algorithm described in Section 2.1 is changed to identify and remember the *cause* of the coloring of an $\wedge$-node n that is colored by F. If n was colored by its sons, then $cause(n)$ is the son that was the first to be colored by F. If n was colored due to a witness, then $cause(n)$ is chosen to be one of its sons which resides on the same SCC and was colored by witness as well. There must exist such a son, otherwise n would be colored by its sons. Note that $cause(n)$ depends on the execution of the coloring algorithm.

Given a game-graph $G_{M\times\varphi}$, for a Kripke structure M and a CTL formula φ, and given its coloring χ and an initial node $n_0 = (s_0, \varphi)$ such that $\chi(n_0) = F$, an annotated counterexample is computed by the algorithm `ComputeCounter` shown in Figure 2. Its result is denoted C. This is a DFS/BFS-like algorithm that starts from n_0 (line 1). When it encounters an $\vee$-node it adds all of its sons to C (line 10), and when it encounters an $\wedge$-node it adds its *cause* to C (line 13). Note, that `ComputeCounter` constructs C by adding edges to it, with the meaning that C consists of the corresponding nodes connected by these edges.

```
Function ComputeCounter (G, n_0)
 1   C := new := {n_0}
 2   old := ∅
 3   while new ≠ ∅ do
 4      n := remove (new)
 5      add n to old
 6      if n is a terminal node then continue  \* sons = ∅ *\
 7      if n is an ∨-node then
 8         for each son n' of n in G do
 9            if n' not in old then add n' to new
10            add the edge (n, n') to C
11      if n is an ∧-node then
12         if cause(n) not in old then add cause(n) to new
13         add the edge (n, cause(n)) to C
14   return C
```

Fig. 2. The algorithm `ComputeCounter` for computing an annotated counterexample

Complexity. Clearly, the construction of the annotated counterexample has a linear running time in the size of its result. The result is linear (in the worst case) with respect to the size of the game-graph $G_{M\times\varphi}$. The latter is bounded by the size of the underlying Kripke structure times the length of the CTL formula, i.e. $O(|M| \cdot |\varphi|)$.

Intuitively speaking, the result of `ComputeCounter` is indeed a counterexample in the sense that it points out the reasons for φ's refutation on the model. Each node in the subgraph C is marked by a state s and by a subformula φ_1, such that $\chi((s,\varphi_1)) = F$ (as claimed by Lemma 3.7), thus by Theorem 2.9, $[s \models \varphi_1] = \text{ff}$. The edges point out the reason (cause) for the refutation of a certain subformula in a certain state: the refutation in an $\wedge$-node is shown by refutation in one of its sons, whereas the refutation in an $\vee$-node is shown by all its sons.

Before proving that the result of `ComputeCounter` meets the formal definition of an annotated counterexample, we would like to emphasize that for the correctness of the algorithm and its result, it is *mandatory* to choose for an $\wedge$-node the son that caused the coloring of the node, and not any son that was colored by F. The following example demonstrates its importance.

Example 3.6. Figure 3 presents an example for computing an annotated counterexample using the algorithm `ComputeCounter` and demonstrates the necessity of choosing $cause(n)$ as a son of an $\wedge$-node n when computing an annotated counterexample. Figure 3(a) presents a colored game-graph G for the model M and $\varphi = A(pVq)$, where grey nodes are colored by F, whereas white nodes are colored by T, and bold edges point to the *cause* of an $\wedge$-node. The initial node $(s, A(pVq))$ is colored by F, i.e. $[s \models A(pVq)] = \text{ff}$. Figure 3(b) presents the annotated counterexample computed by `ComputeCounter`, where it can be seen that the reason for refutation is the existence of the path $s, s_1, \ldots$ and particularly its prefix s, s_1, where q is not satisfied by s_1, although it was not "released" by p (p does not hold in s). On the other hand, Figure 3(c) presents a subgraph of G, that is computed by a variation of `ComputeCounter`, where for an $\wedge$-node, an arbitrary son that is colored

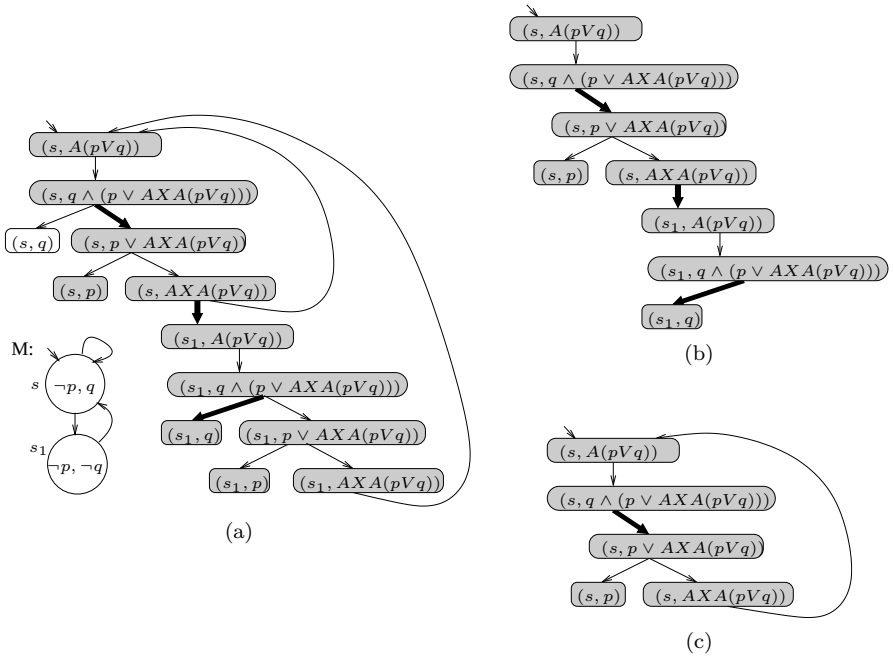

Fig. 3. (a) A colored game-graph for M and $\varphi = A(pVq)$, where bold edges point to the *cause* of an $\wedge$-node, (b) Its annotated counterexample computed by `ComputeCounter` and (c) A possible result of `ComputeCounter` without the use of the *cause* in $\wedge$-nodes.

by F is chosen. In the example, the node $(s, A(pVq))$ was chosen as a refuting son of $(s, AXA(pVq))$ rather than $(s_1, A(pVq))$, which is its cause. The resulting subgraph implies that the refutation of $A(pVq)$ results from the path $s, s, \ldots$. However, this path satisfies pVq, and therefore it does not prove refutation. Thus, this is not a "good" counterexample. More formally, this subgraph can be shown to be *not* independent of G. For example, an initial coloring function χ_I that colors the node $(s_1, A(pVq))$ by T, would result in a partial coloring function of G w.r.t. the subgraph from Figure 3(c), where the nodes $(s, A(pVq))$, $(s, q \wedge (p \vee AXA(pVq)))$, $(s, p \vee AXA(pVq))$ and $(s, AXA(pVq))$ from this subgraph, are colored by T instead of F.

3.2 Properties of the Computed Annotated Counterexample

We postpone the correctness proof of algorithm `ComputeCounter` to Section 3.3. In this section we discuss some interesting properties of subgraphs produced by `ComputeCounter`, some of which are relevant to the correctness proof. Throughout this section we set G to be a game-graph with a coloring function χ, and denote by C the subgraph of G computed by `ComputeCounter`.

LEMMA 3.7. *For each node $n \in C$, we have that $\chi(n) = F$.*

LEMMA 3.8. *(1) If a node in C was colored due to a witness, then it belongs to a non-trivial SCC in C.*

(*2*) *Each non-trivial SCC in C contains at least one node that was colored due to a witness.*

From Lemma 3.7 and Lemma 3.8 we have the following conclusion.

COROLLARY 3.9. *Non-trivial SCCs in C are either AU-SCCs or EU-SCCs.*

The property of C described in Lemma 3.8, along with Corollary 3.9, implies that non-trivial SCCs appear in C iff at least one of their nodes was colored due to an AU or EU witness. That is, any non-trivial SCC that appears in the annotated counterexample indicates a refutation of the U operator, which results, at least partly, from an infinite path, where weak until[3] is satisfied, but not strong until (which is used in our work). This intuition results from the properties of the coloring algorithm. If a node is colored due to a witness, this means that finite information alone is not sufficient to cause its color. More specifically, in the case of $A(\varphi_1 U \varphi_2)$, this means that there is no finite (prefix of a) path where φ_1 ceases being satisfied before φ_2 is satisfied, and the refutation results from an infinite path where φ_1 is always satisfied, but φ_2 is never satisfied. In case of $E(\varphi_1 U \varphi_2)$, this means that the refutation results, at least partly, from infinite evidence of this form and not only from finite (prefixes of) paths.

Since the algorithm `ComputeCounter` is designed to find counterexamples for full CTL, and in particular for existential properties, its result C has a more complex structure than counterexamples that are used for universal properties. Yet, the following lemma shows that when applied to formulae in ACTL, where only universal properties exist, the result of `ComputeCounter` has a simpler structure. In fact, it has a *tree-like* structure, as defined in [Clarke et al. 2002]. It differs from the counterexamples presented in [Clarke et al. 2002] only in the existence of annotations.

LEMMA 3.10. *Non-trivial AU-SCCs in C are always simple cycles, rather than general SCCs.*

3.3 Correctness of the Computed Annotated Counterexample

We now formally show that the result of algorithm `ComputeCounter` is indeed an annotated counterexample, as defined in Definition 3.5. We stick to the notations of Section 3.2 and refer to the game-graph on which `Computecounter` was applied as G, to its coloring function as χ, and to the subgraph that it produced by `ComputeCounter` as C.

The first requirement of Definition 3.5 is obviously fulfilled, as described by Lemma 3.7. The following theorems state that C satisfies the other two requirements as well.

THEOREM 3.11. *The subgraph C, computed by* `ComputeCounter`*, is independent of the game-graph G.*

The correctness of Theorem 3.11 strongly depends on the choice of $cause(n)$ as the son of an $\wedge$-node in the algorithm `ComputeCounter`, as demonstrated by Example 3.6. The following theorem refers to the minimality of C.

[3]The weak version of the until operator, $\varphi_1 W \varphi_2$, does not guarantee that φ_2 holds eventually.

THEOREM 3.12. *C is* minimal *w.r.t. the property of being independent of the game-graph G.*

COROLLARY 3.13. *Let G be a game-graph with an initial node colored F. Then the subgraph C of G, computed by* `ComputeCounter`*, is an annotated counterexample.*

3.4 Practical Considerations

Since the computed annotated counterexample C may be big and difficult to understand, several simplifications may be suggested in order to make it smaller, and thus easier to navigate and to comprehend.

Zoom In - Zoom Out. Since each non-trivial SCC in C is "attached" to a single *AU* or *EU* formula, as indicated by Corollary 3.9, the annotated counterexample can be "zoomed-out" into a DAG. This can be done by constructing the (maximal strongly connected) components graph of the annotated counterexample, where each non-trivial MSCC is replaced by a single node, annotated with its witness. This way, we allow a "global" view on the annotated counterexample, along with the possibility to "zoom in" into each MSCC and view its inner structure. This allows a user to interact with the system and navigate through different parts of the annotated counterexample.

Reductions of the Annotated Counterexample. The complexity of the annotated counterexample results, at least partly, from the node annotations and the auxiliary edges, that add information about the formula. This auxiliary information is valuable in the sense that it helps in understanding the counterexample. However, the annotated counterexample can be "reduced" into more compact structures by hiding (some of) the information that results from the formula. Auxiliary edges may be collapsed by merging nodes that were originally separated by them, resulting in a subgraph of the unwound model. Node annotations can either be removed or partially remembered. These simplifications reflect the trade-off between the size of the counter-example and the additional information originating from the formula.

4. GAME-BASED MODEL CHECKING ON ABSTRACT MODELS

In this section, we extend the discussion to abstract models. We suggest a generalization of the game-based model checking algorithm for evaluating a CTL formula φ over a KMTS M (that represents an abstract model) according to the 3-valued semantics. In Appendix A we consider the abstract *2-valued semantics* of CTL, which is also suitable for KMTSs. We present an abstract model checking algorithm based on the 2-valued semantics and discuss solving the 3-valued problem by reducing it to two instances of the 2-valued problem, as suggested in [Bruns and Godefroid 2000][4]. We also discuss the advantages of the direct solution, described in this section.

We start with the description of the 3-valued game. The 3-valued game is still played by ∃loise and ∀belard. The game *board* and the form of a *play* are defined

[4]In [Bruns and Godefroid 2000] the reduction has been suggested for partial Kripke structures. Yet, [Godefroid and Jagadeesan 2003] extends this approach to KMTSs as well.

as in the "concrete" game described in Section 2.1. However, the moves of the players and the winning criteria need to be adapted. The main difference arises from the fact that KMTSs have two types of transitions. Since the transitions of the model are considered only in configurations with subformulae of the form $AX\varphi_1$ or $EX\varphi_1$, these are the only cases where the rules of the game need to be changed. Intuitively, in order to be able to both prove and refute each subformula, the game needs to allow the players to use both may and must transitions in such configurations. The reason is that, for example, truth of a formula $AX\varphi_1$ in a state s should be checked upon outgoing may-transitions of s, but its falseness should be checked upon outgoing must-transitions. Therefore, the new moves of the game consist of the moves of the original game (see Section 2.1), with the exception that moves (2) and (3) are adapted as follows.

The new moves of the game:

(2) if $C_i = (s, AX\varphi)$, then ∀belard chooses a transition $s \xrightarrow{\text{must}} s'$ (for refutation) or $s \xrightarrow{\text{may}} s'$ (to prevent satisfaction), and $C_{i+1} = (s', \varphi)$.

(3) if $C_i = (s, EX\varphi)$, then ∃loise chooses a transition $s \xrightarrow{\text{must}} s'$ (for satisfaction) or $s \xrightarrow{\text{may}} s'$ (to prevent refutation), and $C_{i+1} = (s', \varphi)$.

That is, each player can use both may and must transitions. Intuitively, the players use must-transitions in order to win, while they use may-transitions in order to prevent the other player from winning. To demonstrate this, consider again the $AX\varphi_1$ example: in a configuration of the form $(s, AX\varphi_1)$, ∀belard makes a move. If he wants to win he needs to refute $AX\varphi_1$. Thus he needs to show a *must*-transition from s to a state that refutes φ_1. Yet, if he only wishes to prevent ∃loise from winning, then it suffices to show a *may*-transition to a state that does not satisfy φ_1. As a result it is possible that none of the players wins the play, i.e. the play ends with a *tie*. As before, a *maximal* play, defined in Section 2.1, is infinite if and only if exactly one *witness*, which is either an AU,EU,AV or EV-formula, appears in it infinitely often. However, the winning rules become more complicated. A player can only win the play if he or she is "consistent" in their moves, meaning that the player always makes moves that are designed for satisfaction (if the player is ∃loise), or always makes moves that are designed for refutation (if it is ∀belard). These moves are all based on must transitions. The other player, on the other hand, possibly uses both types of transitions.

Definition 4.1. (1) A play is called *true-consistent* if in each configuration of the form $C_i = (s, EX\varphi)$, ∃loise chooses a move based on $\xrightarrow{\text{must}}$ transitions.

(2) A play is called *false-consistent* if in each configuration of the form $C_i = (s, AX\varphi)$, ∀belard chooses a move based on $\xrightarrow{\text{must}}$ transitions.

The new winning criteria:

—∀belard wins the play iff the play is false-consistent and in addition one of the following holds:

(1) the play is finite and ends in a terminal configuration of the form $C_i = (s, \text{false})$, or $C_i = (s, l)$, where $\neg l \in L(s)$.

(2) the play is infinite and the witness is of the form AU or EU.

—∃loise wins the play iff the play is true-consistent and in addition one of the following holds:
(1) the play is finite and ends in a terminal configuration of the form $C_i = (s, \text{true})$, or $C_i = (s, l)$, where $l \in L(s)$.
(2) the play is infinite and the witness is of the form AV or EV.

—Otherwise, the play ends with a tie.

Strategies and winning strategies are defined as before. We now have the following correspondence between the game and the truth value of a formula in a certain state under the 3-valued semantics.

THEOREM 4.2. *Let M be a KMTS and φ a CTL formula. Then, for each $s \in S$:*

(1) $[(M,s) \models^3 \varphi] = tt$ *iff* *∃loise has a winning strategy starting at* (s,φ).

(2) $[(M,s) \models^3 \varphi] = ff$ *iff* *∀belard has a winning strategy starting at* (s,φ).

(3) $[(M,s) \models^3 \varphi] = \bot$ *iff* *none of the players has a winning strategy starting at* (s,φ).

PROOF. The proof is by induction on the structure of CTL formulae. It suffices to prove the implication from the truth value to the existence (or non-existence) of a winning strategy. We do that by construction of strategies for the players.

Base case: φ = true, false or l, where $l \in Lit$. Thus, $C_0 = (s, \varphi)$ is a terminal configuration. By the winning criteria, if $[s \models^3 \varphi] = \text{tt}$, then ∃loise wins every play, right at the beginning. If $[s \models^3 \varphi] = \text{ff}$, then ∀belard wins every play, right at the beginning. Otherwise, every play ends with a tie, and therefore none of them has a winning strategy.

Induction step:

—$\varphi = \varphi_1 \vee \varphi_2$:
(1) If $[s \models^3 \varphi] = \text{tt}$, then by the definition of the 3-valued semantics, there exists $j \in \{1,2\}$ such that $[s \models^3 \varphi_j] = \text{tt}$. By the induction hypothesis, ∃loise has a winning strategy for every play that starts from (s, φ_j). Thus, her winning strategy, starting from (s,φ), consists of choosing the next move, which is her responsibility, to be (s, φ_j) and from then on, using the guaranteed winning strategy for (s, φ_j).
(2) If $[s \models^3 \varphi] = \text{ff}$, then by the semantics definition, for each $j \in \{1,2\}$ we have that $[s \models^3 \varphi_j] = \text{ff}$. By the induction hypothesis, ∀belard has a winning strategy for every play that starts from either one of the configurations (s, φ_j). The union of these winning strategies is his winning strategy for the game starting from (s,φ). This is indeed a winning strategy, because no matter which of the two possible configurations is chosen by ∃loise as her move, ∀belard has a winning strategy from this point and on.
(3) If $[s \models^3 \varphi] = \bot$, then by the definition, at least for one of $j \in \{1,2\}$ we have that $[s \models^3 \varphi_j] = \bot$, and for the other one, $k \in \{1,2\}$, $k \neq j$, we have that the

value of $[s \models^3 \varphi_k]$ is $\perp$ or ff. Thus, $\forall$belard has no winning strategy because $\exists$loise, that makes the move in the initial configuration, can always choose to proceed to (s, φ_j), for which $\forall$belard has no winning strategy, by the induction hypothesis. In addition, $\exists$loise has no winning strategy, because no matter which move she makes, she reaches a configuration for which by the induction hypothesis she does not have a winning strategy.

—$\varphi = \varphi_1 \wedge \varphi_2$: symmetric, since the definition of 3-valued semantics is opposite and so are the roles of the players in a configuration with such a formula.

—$\varphi = EX\varphi_1$:

(1) If $[s \models^3 \varphi] = \text{tt}$, then by the semantics definition, there exists a transition $s \xrightarrow{\text{must}} s'$, such that $[s' \models^3 \varphi_1] = \text{tt}$. By the induction hypothesis, $\exists$loise has a winning strategy for every play that starts from (s', φ_1). Thus, her winning strategy, starting from (s, φ), consists of choosing the next move, which is her responsibility, to be (s', φ_1), which allows her to maintain the play true-consistent, and from then on, using the guaranteed winning strategy for (s, φ_j).

(2) If $[s \models^3 \varphi] = \text{ff}$, then by the semantics definition, for each transition $s \xrightarrow{\text{may}} s'$, we have that $[s' \models^3 \varphi_1] = \text{ff}$. Recall that $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$, thus this applies to $\xrightarrow{\text{must}}$ transitions as well. By the induction hypothesis, $\forall$belard has a winning strategy for every play that starts from either one of these configurations (s', φ_1). The union of these winning strategies is his winning strategy for the game starting from (s, φ). This is indeed a winning strategy, because these are the only possibilities that $\exists$loise has for the first move, and no matter which of the possible configurations is chosen by her, $\forall$belard has a winning strategy from this point and on.

(3) If $[s \models^3 \varphi] = \perp$, then by the definition, there exists at least one transition $s \xrightarrow{\text{may}} s'$ such that the value of $[s' \models^3 \varphi_1]$ is $\perp$ or tt, and for all the transitions $s \xrightarrow{\text{must}} s''$, we have that the value of $[s'' \models^3 \varphi_1]$ is $\perp$ or ff. Thus, $\forall$belard has no winning strategy because $\exists$loise, that makes the move in the initial configuration, can always choose to proceed to (s', φ_1), for which $\forall$belard has no winning strategy, by the induction hypothesis. In addition, $\exists$loise has no winning strategy, because for her to win, the play must be true-consistent, and thus she has to choose as her first move one of the configurations (s'', φ_1), which are based on must-transitions. However, no matter which of them she chooses, she reaches a configuration for which by the induction hypothesis she does not have a winning strategy.

—$\varphi = AX\varphi_1$: symmetric, since the definition is opposite and so are the roles of the players.

—$\varphi = A(\varphi_1 U \varphi_2)$:

First note that (the prefix of) each play that starts from the configuration (s, φ) is of the *periodic* form $(s, A(\varphi_1 U \varphi_2)) \rightarrow (s, \varphi_2 \vee (\varphi_1 \wedge AXA(\varphi_1 U \varphi_2))) \rightarrow_\exists (s, \varphi_1 \wedge AXA(\varphi_1 U \varphi_2)) \rightarrow_\forall (s, AXA(\varphi_1 U \varphi_2)) \rightarrow_\forall (s_1, A(\varphi_1 U \varphi_2)) \rightarrow \dots$, based on some (must or may) path $s, s_1, \dots$ from s. This form continues as long as none of the players chooses as a next move (s_i, φ_2) ($\exists$loise) or (s_i, φ_1) ($\forall$belard)

from the configurations $(s_i, \varphi_2 \vee (\varphi_1 \wedge AXA(\varphi_1 U \varphi_2)))$ or $(s_i, \varphi_1 \wedge AXA(\varphi_1 U \varphi_2))$ respectively.

If a player chooses as a next move (s_i, φ_1) or (s_i, φ_2), we say that he or she *interrupts* the periodic form of the play in index i. Otherwise, we say that he or she *maintains* the periodic form of the play in index i.

In addition note that in fact ∀belard is responsible for choosing the path that the play is based on, because in configurations of the form $(s_i, AXA(\varphi_1 U \varphi_2))$ where the move is based on a transition of the model, ∀belard is the player that makes the move. If ∀belard bases his moves on a path $\pi = s_0, s_1, \ldots$ and maintains the periodic form of the play in his move in index i, then this means that he continues from $(s_i, \varphi_1 \wedge AXA(\varphi_1 U \varphi_2))$ to $(s_i, AXA(\varphi_1 U \varphi_2))$ and to $(s_{i+1}, A(\varphi_1 U \varphi_2))$. In this case we say that ∀belard *proceeds along* π in index i.

(1) If $[s \models^3 \varphi] = \text{tt}$, then by the semantics definition, for each may-path $\pi = s_0, s_1, \ldots$ from s, we have that $[\pi \models^3 \varphi_1 U \varphi_2] = \text{tt}$. This means that for each such path there exists $0 \leq k < |\pi|$ such that $[s_k \models^3 \varphi_2] = \text{tt}$ and for all $j < k$: $[s_j \models^3 \varphi_1] = \text{tt}$. Thus, by the induction hypothesis, ∃loise has a winning strategy for each game that starts from either one of the corresponding configurations (s_k, φ_2) and (s_j, φ_1). The winning strategy of ∃loise for a game that starts from the configuration (s, φ) consists of all these winning strategies, along with the rules that tell her to interrupt the periodic form of the play and proceed to (s_i, φ_2) if $i = k$ for some k as described above, and to maintain the periodic form if $i < k$. This is a winning strategy, because as long as $i < k$ for some k as described above, if ∀belard chooses (s_i, φ_1), then since $[s_i \models^3 \varphi_1] = \text{tt}$, by the induction hypothesis ∃loise has a winning strategy. Otherwise, ∀belard maintains the periodic form of the play in such i's, and so does ∃loise (as described by her strategy). When $i = k$ is reached, ∃loise chooses the configuration (s_k, φ_2), for which she has a winning strategy by the induction hypothesis, since $[s_k \models^3 \varphi_2] = \text{tt}$.

(2) If $[s \models^3 \varphi] = \text{ff}$, then by its definition, there exists a must-path $\pi = s_0, s_1, \ldots$ from s, such that $[\pi \models^3 \varphi_1 U \varphi_2] = \text{ff}$. This means that $\big(\forall\, 0 \leq k < |\pi| : [(\forall j < k : [s_j \models^3 \varphi_1] \neq \text{ff}) \Rightarrow ([s_k \models^3 \varphi_2] = \text{ff})]\big) \wedge \big((\forall\, 0 \leq k < |\pi| : [s_k \models^3 \varphi_1] \neq \text{ff}) \Rightarrow |\pi| = \infty\big)$. If there exists $0 \leq i < |\pi|$ for which $[s_i \models^3 \varphi_1] = \text{ff}$, then let k be the *smallest* index for which $[s_k \models^3 \varphi_1] = \text{ff}$. Otherwise, we set $k = \infty$. Note that in this case $|\pi| = \infty$ as well. Either way, by the minimality of k, we know that for every $i < k + 1$ the formula $\forall j < i : [s_j \models^3 \varphi_1] \neq \text{ff}$ holds, which implies that for every $i < k + 1$: $[s_i \models^3 \varphi_2] = \text{ff}$.

By the induction hypothesis, ∀belard has a winning strategy from every configuration (s_i, φ_2) for which $[s_i \models^3 \varphi_2] = \text{ff}$ and from every configuration (s_i, φ_1) for which $[s_i \models^3 \varphi_1] = \text{ff}$. Thus, his winning strategy for a game that starts from the configuration (s, φ) includes these winning strategies, as well as rules that tell him to interrupt the periodic form of the play and proceed to (s_i, φ_1) if $i = k$ (of course this is only possible if k is finite), and to proceed along π otherwise. This is a winning strategy, because as long as $i < k + 1$,

if ∃loise chooses (s_i, φ_2), then by the induction hypothesis ∀belard has a winning strategy (which is part of his winning strategy from (s, φ)), since $[s_i \overset{3}{\models} \varphi_2] =$ ff. Otherwise, ∃loise maintains the periodic form of the play in such i's. As for ∀belard, as long as $i < k$, he proceeds along the must-path π, maintaining the play false-consistent. Now, there are two possibilities:

(a) If k is finite, then when $i = k$ is reached, by the described strategy ∀belard chooses the configuration (s_k, φ_1) as his next move. Since by the choice of k $[s_k \overset{3}{\models} \varphi_1] =$ ff, then by the induction hypothesis ∀belard has a winning strategy from this configuration.

(b) Otherwise, we have that the must-path π itself is infinite and for every $i \geq 0$ ∀belard *proceeds along* π, such that the play is infinite and false-consistent with the AU-witness. Thus, ∀belard wins as well.

(3) If $[s \overset{3}{\models} \varphi] = \bot$, then by the semantics definition, there exists a may-path π from s, for which the value of $[\pi \overset{3}{\models} \varphi_1 U \varphi_2]$ is $\bot$ or ff, and in addition for each must-path π' the value of $[\pi' \overset{3}{\models} \varphi_1 U \varphi_2]$ is $\bot$ or tt. The outline of the proof is as follows. The existence of π allows ∀belard to prevent ∃loise from winning, which shows that ∃loise does not have a winning strategy. On the other hand, the property of each must-path shows that ∀belard does not have a winning strategy. The details of the proof are similar to the previous cases and are thus omitted[5].

—$\varphi = E(\varphi_1 U \varphi_2)$, $A(\varphi_1 V \varphi_2)$ or $E(\varphi_1 V \varphi_2)$: similar.

□

Theorem 4.2 refers to the existence (or non-existence) of winning strategies for the players. One may also be interested in the existence of *memoryless* winning strategies. The interested reader is referred to appendix B, where we define the notion of a *memoryless* strategy and it is shown that Theorem 4.2 can be rephrased to refer to *memoryless* winning strategies.

In order to use the correspondence described in Theorem 4.2 for model checking, we generalize the game-based model checking algorithm.

4.1 Game-Graph Construction and its Properties

The construction of the (3-valued) game-graph, denoted $G_{M \times \varphi}$, is defined as for the "concrete" game (as described in Section 2.1). The nodes of the game-graph, denoted N, can again be classified as $\wedge$-nodes, $\vee$-nodes, AX-nodes and EX-nodes. Similarly, the edges can be classified as *progress* edges, that originate in AX or EX nodes, or *auxiliary* edges. But now, we distinguish between two types of progress edges, two types of sons and two types of SCCs.

—Edges that are based on must-transitions are referred to as *must-edges*. Edges that are based on may-transitions are referred to as *may-edges*.

—A node n' is a *may-son* of the node n if there exists a may-edge (n, n'). n' is a *must-son* of n if there exists a must-edge (n, n').

[5]The full proof can be found in [Shoham 2003].

—An SCC in the game-graph is a *may-SCC* if all its progress edges are may-edges. It is a *must-SCC* if all its progress edges are must-edges.

4.2 Coloring Algorithm

The coloring algorithm of the 3-valued game-graph needs to be adapted as well. First, a new color, denoted ?, is introduced for configurations in which none of the players has a winning strategy.

Second, the partition to Q_i's that is based on MSCCs is affected because there are two types of SCCs (and MSCCs) in $G_{M\times\varphi}$. However, $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$, thus each must-edge is also a may-edge and every must-SCC is also a may-SCC. As a result, the graph can be partitioned to may-MSCCs (based on the may-edges). Note that Lemma 2.6 holds for may-SCCs in the 3-valued game-graph as well. Thus, the notion of a *witness* in an SCC is also valid.

As for the coloring itself, similarly to the concrete case, the 3-valued coloring algorithm processes the game-graph bottom-up. The rules that color a node as a terminal node or based on its sons are adapted to use the 3 colors and to reflect the 3-valued semantics of $\wedge$, $\vee$, AX and EX. Once again, if at any point the coloring cannot proceed, then the existence of a non-trivial SCC and a corresponding witness is implied. Yet, in this case further analysis is needed before being able to color the remaining nodes. If the witness is of the form AU, then uncolored loops can only be used to prove its refutation. To do so, "real" loops are needed. Thus for such a witness, we need to have an uncolored non-trivial *must-SCC* in order to color it by a definite color (F). On the other hand, for an AV-witness, loops can contribute to satisfaction, and satisfaction of universal properties should be examined upon may-transitions. Thus for such a witness, *may-edges* are sufficient to color the non-trivial SCC by T. Similarly, for an EU witness, a may-SCC suffices, whereas for an EV witness, a must-SCC is used. Thus, in the 3-valued case, when the use of a witness is needed, we first apply a special method whose purpose is to reduce the uncolored non-trivial SCC to one with the type required for a definite color. The nodes that are removed during this process are colored ?, and only the remaining nodes are colored by a definite color (depending on the witness).

The (3-valued) coloring algorithm

Partition and Order: The game-graph is partitioned into its may-MSCCs. The resulting sets are denoted Q_i's. This partition induces a partial order such that transitions go out of a set only to itself or to a "smaller" set. The partial order is extended to a total order $\leq$ arbitrarily.

Coloring: As before, the coloring algorithm processes the Q_i's according to $\leq$, bottom-up. Let Q_i be the smallest set w.r.t. $\leq$ that is not yet fully colored. The nodes of Q_i are colored in two phases, as follows.

Phase 1: *Sons-coloring phase:*

Apply the following rules to all the nodes in Q_i until none of them is applicable.

—A terminal node is colored by T if $\exists$loise wins in it, by F if $\forall$belard wins in it, and by ? otherwise.

—An AX-node is colored by:

—T if all its may-sons are colored by T.

—F if it has a must-son that is colored by F.
—? if all its must-sons are colored by T or ? and it has a may-son that is colored by F or ?.

—An EX-node is colored by:
—T if it has a must-son that is colored by T.
—F if all its may-sons are colored by F.
—? if it has a may-son that is colored by T or ? and all its must-sons are colored by F or ?.

—An $\wedge$-node, other than AX-node, is colored by:
—T if both its sons are colored by T.
—F if it has a son that is colored by F.
—? if it has a son that is colored by ? and the other one is colored by ? or T.

—An $\vee$-node, other than EX-node, is colored by:
—T if it has a son that is colored by T.
—F if both its sons are colored by F.
—? if it has a son that is colored by ? and the other one is colored by ? or F.

Note that auxiliary edges can be considered as both may and must edges. By doing so, it is possible to join the description of the coloring of AX-nodes and other $\wedge$-nodes, as well as the description of EX-nodes and other $\vee$-nodes. We do not do that for clarity.

Phase 2: *Witness-coloring phase:*

If after the propagation of the rules of phase 1 there are still nodes in Q_i that remain uncolored, then Q_i must be a non-trivial may-MSCC that has exactly one witness (by Lemma 2.6 which holds here as well). The uncolored nodes in Q_i are colored in two phases, according to the type of the witness. We consider two cases.

Case U: The witness is of the form $A(\varphi_1 U \varphi_2)$ or $E(\varphi_1 U \varphi_2)$:

Phase 2a: Repeatedly color by ? each node in Q_i that satisfies one of the following conditions, until there is no change (i.e. none of the conditions holds for any node in Q_i).

—An AX-node that all its must sons are colored by T or ?.
—An $\wedge$-node that both its sons are colored by T or ?.
—An EX-node that has a may-son that is colored by T or ?.
—An $\vee$-node that has a son that is colored by T or ?. (or equivalently: An $\vee$-node that has a son that is colored by ?, since the T option is impossible).

In fact, each node for which the F option is no longer possible according to the rules of phase 1 (the *sons-coloring* phase) is colored by ?.

Phase 2b: Color the remaining nodes in Q_i by F.

Case V: The witness is of the form $A(\varphi_1 V \varphi_2)$ or $E(\varphi_1 V \varphi_2)$:

Phase 2a: Repeatedly color by ? each node in Q_i that satisfies one of the following conditions, until there is no change (i.e. none of the conditions holds for any node in Q_i).

—An AX-node that has a may son that is colored by F or ?.

—An $\wedge$-node that has a son that is colored by F or ?. (or equivalently: An $\wedge$-node that has a son that is colored by ?, since the F option is impossible).
—An EX-node that all its must sons are colored by F or ?.
—An $\vee$-node that both its sons are colored by F or ?.
In fact, each node for which the T option is no longer possible according to the rules of phase 1 (the *sons-coloring* phase) is colored by ?.

Phase 2b: Color the remaining nodes in Q_i by T.

The result of the coloring algorithm is a *3-valued coloring function* $\chi : N \rightarrow \{T, F, ?\}$.

Note that coloring by ? is done carefully. One may suggest to color a node by ? in any case where it is not colored and the coloring can not proceed. However, since we would like to follow the 3-valued semantics, we need to color such a node by ? only if it is *not possible* that it should be colored by T or F. This is not implied by the node being uncolored. Therefore, a node is colored by ? only if there is *evidence* that it cannot be colored otherwise. In any other case, another method is used to determine its color. The correctness of the coloring algorithm is guaranteed by the following theorems.

THEOREM 4.3. *All the nodes in the game-graph get colored by the 3-valued coloring algorithm.*

THEOREM 4.4. *Let $G_{M\times\varphi}$ be a 3-valued game-graph, colored by the above 3-valued coloring algorithm with χ being the coloring function, and let n be a node in the game-graph, then:*

(1) $\chi(n) = T$ *iff* $\exists$*loise has a winning strategy starting at* n.
(2) $\chi(n) = F$ *iff* $\forall$*belard has a winning strategy starting at* n.
(3) $\chi(n) =?$ *iff none of the players has a winning strategy starting at* n.

The correctness of Theorems 4.3 and 4.4 strongly depends on the observations described by the following lemmas.

LEMMA 4.5. *Let n be a node that is uncolored at the beginning of phase 2 in its set Q_i. Then n lies on a non-trivial SCC that is a subgraph of Q_i and all nodes of the SCC are uncolored at the beginning of phase 2.*

We conclude that if a set Q_i has uncolored nodes at the beginning of phase 2 then Q_i is a non-trivial *may-MSCC.*

LEMMA 4.6. *Let Q_i be the set that is handled by the coloring algorithm at its ith iteration, and let n be a node in Q_i that is uncolored at the beginning of phase 2b. Then*

(1) If Q_i has an EU or AV witness, then n lies on a non-trivial may-SCC *that is a subgraph of Q_i and all nodes of the may-SCC are uncolored at the beginning of phase 2b.*
(2) If Q_i has an AU or EV witness, then n lies on a non-trivial must-SCC *that is a subgraph of Q_i and all nodes of the must-SCC are uncolored at the beginning of phase 2b.*

Implementation Issues. First, note that the correctness of the coloring algorithm is not damaged if during phase 1 of the ith iteration, nodes from sets other than Q_i are colored as well. This results from the property that once a node is colored, its color never changes. In other words, coloring successors of an already colored node does not change its color. Thus, if a node from a set Q_j can be colored in phase 1 of the ith iteration ($i < j$) based on its sons, then these sons will still be colored the same in the jth iteration, leading to the same coloring (although possibly additional sons will be colored in the jth iteration).

Based on this observation, the coloring algorithm can be implemented in linear running time, using an AND/OR graph, similarly to the algorithm described in [Kupferman et al. 2000] for checking the nonemptiness of the language of a simple weak alternating word automaton. The algorithm maintains an integer i that contains the current iteration and three stacks S_T, S_F and $S_?$. The stacks contain nodes that were colored by T, F or ?, yet still have not propagated their coloring further. At the beginning, the stacks are initialized by all the terminal nodes, according to their coloring. During the son-based coloring (phase 1), whenever a node is colored, it is pushed into one of the stacks, based on its color. As long as the stacks are not empty, a node is popped from one of them and its parents are checked to see if they can be colored (based on *all* their sons).

When the stacks are empty, the witness-based coloring (phase 2) is applied on the smallest Q_i that contains uncolored nodes. First, phase 2a is applied, where initially all the nodes in Q_i are checked once to see if they can be colored. The ones that are colored are pushed into $S_?$ as well as a temporary stack. The temporary stack is used to propagate their coloring within Q_i based on the rules of phase 2a and each node that gets colored is added to the temporary stack and to $S_?$. Phase 2a ends when the temporary stack is empty. At that time, all the remaining nodes in Q_i are colored by T or F (depending on the witness) in phase 2b and are pushed into the appropriate stack. The algorithm then returns to phase 1 (with the new content of the stacks).

Complexity. The running time of the coloring algorithm is linear with respect to the size of the game-graph $G_{M\times\varphi}$. The latter is again bounded by the size of the underlying KMTS times the length of the CTL formula, i.e. $O(|M| \cdot |\varphi|)$.

As a conclusion of Theorem 4.2 and Theorem 4.4, we get the following theorem.

THEOREM 4.7. *Let M be a KMTS and φ a CTL formula. Then, for each $n = (s, \varphi_1) \in G_{M\times\varphi}$:*

(1) $[(M,s) \stackrel{3}{\models} \varphi_1] = tt$ *iff* $n = (s,\varphi_1)$ *is colored by* T.

(2) $[(M,s) \stackrel{3}{\models} \varphi_1] = ff$ *iff* $n = (s,\varphi_1)$ *is colored by* F.

(3) $[(M,s) \stackrel{3}{\models} \varphi_1] = \perp$ *iff* $n = (s,\varphi_1)$ *is colored by* ?.

Given the colored game-graph, if all the initial nodes are colored by T, or if at least one of them is colored by F, then by Theorem 4.7 along with Theorem 2.16, there is a definite answer as for the satisfaction of φ in the *concrete* model. This is because there exists a mixed simulation from the concrete model to the abstract one. Furthermore, if the result is ff, a *concrete* annotated counterexample can be

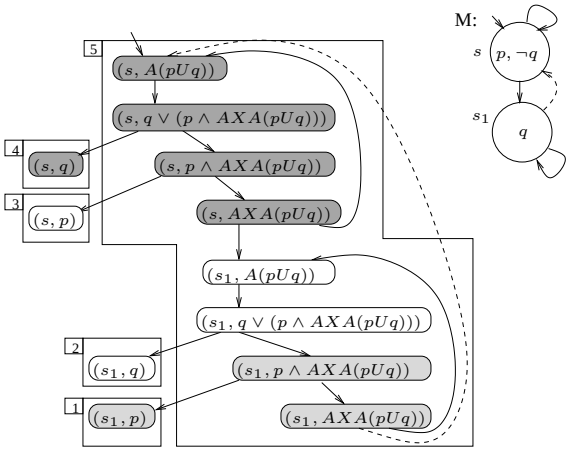

Fig. 4. A colored 3-valued game-graph for M and $\varphi = A(pUq)$, where dashed edges are may-edges, solid edges are must-edges or auxiliary edges, and rectangles depict the partition of the nodes. White nodes are colored by T, dark grey nodes are colored by F, and light grey nodes are colored by ?.

produced, using an extension of the `ComputeCounter` algorithm as described in the following section.

Example 4.8. Figure 4 presents a 3-valued game-graph G and its coloring, where dashed edges represent may-edges and solid edges represent must-edges, as well as auxiliary edges. The partition of G to Q_i's (may-MSCCs) is depicted by rectangles in Figure 4, where their indices 1-5 determine the order $\leq$. The coloring algorithm starts from Q_1-Q_4, where the terminal nodes are colored in phase 1. Note, that neither p nor $\neg p$ label s_1, thus (s_1, p) is colored ?. The algorithm then handles Q_5. $(s_1, q \vee (p \wedge AXA(pUq)))$ is colored T due to its son (s_1, q), causing $(s_1, A(pUq))$ to be colored T as well. At this point, none of the remaining nodes can be colored in phase 1. Thus, the algorithm proceeds to phase 2, with Q_5 containing an AU-witness. The node $(s_1, AXA(pUq))$ is colored ? in phase 2a, since it is an AX node and its only must son is colored T. It causes $(s_1, p \wedge AXA(pUq))$ to be colored ? as well in the same phase. The rest of the nodes are then colored F in phase 2b. This example demonstrates that if a node is left uncolored after phase 2a in a set with an AU-witness, then it lies on a non-trivial must-SCC that provides an evidence for refutation. The final coloring function can be seen in Figure 4, where white nodes are colored T, dark grey nodes are colored F and light grey nodes are colored ?. Based on Theorem 4.7, and since the initial node $(s, A(pUq))$ is colored by F, we can conclude that the value of the formula $\varphi = A(pUq)$ in the initial state s of the model M (and thus in M in general) is ff. This is indeed correct since the value of the until formula pUq in the (infinite) must-path $s, s, \ldots$ is ff (the value of q in s is ff, refuting the eventuality of the until). This implies that φ is refuted in any concrete model that is represented by M.

5. *CONCRETE* ANNOTATED COUNTEREXAMPLES BASED ON *ABSTRACT* GAME-GRAPHS

In this section we show how to produce a *concrete* annotated counterexample from the 3-valued abstract game-graph that was used for model checking, in case that one of the initial nodes was colored by F, meaning that $[M \models \varphi] = \text{ff}$.

Let $M_C = (S_C, S_{0C}, \rightarrow, L_C)$ be a (concrete) Kripke structure and let $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$ be a (abstract) KMTS as described in the preliminaries, such that $M_C \preceq M_A$, where $\preceq$ is the mixed simulation relation. Let $\gamma : S_A \rightarrow 2^{S_C}$ be the concretization function. Given the abstract 3-valued game-graph G_A, based on φ and the abstract model M_A, and its coloring function $\chi : N_A \rightarrow \{T, F, ?\}$, such that $\chi(n_{0a}) = F$ for some initial node $n_{0a} = (s_{0a}, \varphi)$, we first use a variation of the algorithm `ComputeCounter`, described in Section 3.1, to produce an abstract annotated counterexample C_A. The difference in the abstract version of the algorithm is that for an AX-node, where the *cause* is added to the annotated counterexample, we now choose the cause to be a *must-son*, and in an EX-node, we add all its *may-sons* to the annotated counterexample. In order to find a concrete annotated counterexample out of C_A, we then need to replace each abstract state s_a in C_A with concrete states from $\gamma(s_a)$.

We use the algorithm `ConcretizeCounter` shown in Figure 5 as a concretization algorithm, that produces a concrete annotated counterexample, denoted C_C, from C_A. Basically, this is a greedy algorithm. It starts by matching the initial abstract node with an initial concrete node (line 1). It then processes pairs of nodes, where each pair consists of a concrete node $n_c = (s_c, \varphi')$ of C_C and a matching abstract node $n_a = (s_a, \varphi')$ from C_A (line 6). For every such pair, the algorithm produces the concrete sons of n_c in C_C based on its type and based on the abstract sons of n_a in C_A. For an AX-node, the annotated counterexample shows one son that refutes the property. Given such a node n_c, we compute its son $n'_c = (s'_c, \varphi_1)$ based on the son $n'_a = (s'_a, \varphi_1)$ of n_a. We replace the state s'_a of n'_a by a concrete state s'_c that is represented by s'_a, while making sure that s_c and s'_c have a concrete transition between them (lines 11,12). This is the only situation where there is some "freedom" in the choice of concrete states. For an EX-node, the annotated counterexample shows refutation in all its sons. Hence, given such a node n_c, the algorithm makes sure to include *all* its concrete sons in the annotated counterexample (lines 15,16). As for other nodes, their sons result from auxiliary edges, that do not represent advancements along transitions of the model. In these cases the algorithm makes sure to attach both the parent and the son with the same state, whereas the formulae in the sons are determined by the sons of n_a (line 22 for $\wedge$-nodes, lines 25,26 for $\vee$-nodes).

Complexity. The running time of the concretization algorithm `ConcretizeCounter` is linear in the size of the concrete annotated counterexample, which is bounded by the size of the concrete Kripke structure M_C times the length of the CTL formula φ, i.e. $O(|M_C| \cdot |\varphi|)$.

LEMMA 5.1. *The algorithm* `ConcretizeCounter` *does not fail. That is, given an abstract annotated counterexample, it always succeeds to produce a result.*

```
Function ConcretizeCounter (C_A, (s_0a, φ), M_C, γ)
 1  choose s_0c ∈ γ(s_0a) ∩ S_0C arbitrarily
 2  C_C := {(s_0c, φ)}                          \* (s_0c, φ) is the initial node of C_C *\
 3  new := {((s_0a, φ), (s_0c, φ))}
 4  old := ∅
 5  while new ≠ ∅ do
 6     (n_c, n_a) := remove (new)
 7     add n_c to old
 8     Let n_c = (s_c, φ'), n_a = (s_a, φ')
 9     if φ' = AXφ_1 then
10        Let (s'_a, φ_1) be the son of n_a in C_A    \* n_a has exactly one son in C_A *\
11        choose s'_c ∈ {s'_c ∈ S_C : s_c → s'_c} ∩ γ(s'_a) arbitrarily
12        add (s'_c, φ_1) to C_C as a son of n_c
13        if (s'_c, φ_1) not in old then add ((s'_c, φ_1), (s'_a, φ_1)) to new
14     if φ' = EXφ_1 then
15        for each state s'_c ∈ S_C such that s_c → s'_c do
16           add (s'_c, φ_1) to C_C as a son of n_c
17           if (s'_c, φ_1) not in old then
18              find s'_a such that s'_c ∈ γ(s'_a)  \* (s'_a, φ_1) is a may son of n_a in C_A *\
19              add ((s'_c, φ_1), (s'_a, φ_1)) to new
20     if φ' = φ_1 ∧ φ_2 then
21        Let (s_a, φ_i), where i ∈ {1, 2}, be the son of n_a in C_A
                                                \* n_a has exactly one son in C_A *\
22        add (s_c, φ_i) to C_C as a son of n_c
23        if (s_c, φ_i) not in old then add ((s_c, φ_i), (s_a, φ_i)) to new
24     if φ' = φ_1 ∨ φ_2 then
25        for each i ∈ {1, 2} do
26           add (s_c, φ_i) to C_C as a son of n_c
27           if (s_c, φ_i) not in old then add ((s_c, φ_i), (s_a, φ_i)) to new
                                                \* (s_a, φ_i) is a son of n_a in C_A *\
28  return C_C
```

Fig. 5. The algorithm `ConcretizeCounter` for producing a *concrete* annotated counterexample

In order to prove the correctness of the concretization algorithm, we need to show that the result C_C that it produces is indeed a (concrete) annotated counterexample for the concrete game-graph G_C, based on M_C and φ. To do so, we need to show that C_C is a subgraph of G_C, containing an initial node, that is (1) colored F by the coloring function of G_C, (2) independent of G_C, i.e. every partial coloring function of G_C w.r.t. C_C does not change the colors of its nodes, and (3) minimal w.r.t. 1 and 2.

LEMMA 5.2. *Let C_C be the result of* `ConcretizeCounter`*. C_C is a subgraph of the concrete game-graph G_C for M_C and φ, containing an initial node.*

As for (1) and (2), it suffices to show that both the regular coloring algorithm and the partial coloring algorithm w.r.t. C_C, given any initial coloring function of $N_C \setminus C_C$, color the nodes of C_C by F. In fact, the regular coloring algorithm can be viewed as an instance of the partial coloring algorithm with respect to C_C where the initial coloring function of $N_C \setminus C_C$ colors each node $n \in N_C \setminus C_C$ as

the (regular) coloring function. Thus, it suffices to prove this claim for the partial coloring algorithm only.

LEMMA 5.3. *Let n_c be a node in the subgraph C_C of the concrete game-graph $G_C = (N_C, E_C)$, computed by* `ConcretizeCounter`*. Then given any initial coloring function of $N_C \setminus C_C$, n_c is colored by F in the (concrete) partial coloring of G_C w.r.t. C_C.*

LEMMA 5.4. *The subgraph C_C, produced by* `ConcreteCounter`*, is minimal w.r.t. the property of being independent of the concrete game-graph G_C.*

We can now conclude the correctness of the concretization algorithm, guaranteed by the following theorem.

THEOREM 5.5. *The subgraph C_C, computed by the algorithm* `ConcretizeCounter`*, is an annotated counterexample for the concrete model M_C and the formula φ.*

6. REFINEMENT

In this section, we show how to exploit the abstract game-graph in order to refine the abstract model in case that the model checking resulted in an indefinite answer.

In the framework of abstraction-refinement, refinement is usually based on a spurious counterexample, and the state in which its concretization fails. This approach is suitable when the model checking is based on 2-valued semantics, where tt is definite, whereas ff is not: when the result is ff, it is still possible that the concrete model satisfies the property. This is why an abstract counterexample may be *spurious* in the sense that it has no matching concrete counterexample. If the concretization of the abstract counterexample fails, meaning that the counterexample does not show refutation in the concrete model, then refinement is applied. The goal of the refinement is then to eliminate the spurious counterexample, hoping to either prove the property or find a real counterexample in the next iteration.

When dealing with the 3-valued semantics, if the result of the model checking is ff, then it is definite as well and no refinement is needed. In this case, refinement is needed when the resulting truth value is indefinite, i.e. $\perp$, in which case there is no reason to assume either one of the definite answers tt or ff. Thus, we would like to base the refinement not on a counterexample as in [Kurshan 1994; Clarke et al. 2000; Barner et al. 2002; Clarke et al. 2002; Chauhan et al. 2002], but on the point(s) that are responsible for the indefinite answer. The goal of the refinement is to discard these points, in the hope of getting a definite result on the refined abstraction.

Let $M_C = (S_C, S_{0C}, \rightarrow, L_C)$ be a concrete Kripke structure and let $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$ be an abstract KMTS as described in the preliminaries such that $M_C \preceq M_A$. Let $\gamma : S_A \rightarrow 2^{S_C}$ be the concretization function. Given the abstract 3-valued game-graph G, based on the abstract model M_A and its coloring function $\chi : N \rightarrow \{T, F, ?\}$, such that $\chi(n_0) =?$ for some initial node n_0, we use the information gained by the coloring algorithm of G in order to refine the abstraction. We refine the abstract model by splitting its abstract states according to criteria obtained by a *failure node*.

Definition 6.1. A node n is a *failure node* if it is colored by ?, whereas none of its sons was colored by ? at the time n got colored by the coloring algorithm.

```
Function FailureSearch (n)
  1  if n is a failure node then return n
  2  else return FailureSearch (cont(n))
```

Fig. 6. The algorithm `FailureSearch` for finding a failure node

Informally, such a node is a failure node in the sense that it can be seen as the point where the loss of information occurred. Thus, it can guide the refinement in hope to avoid the loss of information.

Note that a failure node may have uncolored sons at the time it is colored, some of which may eventually be colored by ?. Also note, that a terminal node that is colored by ? is also considered a failure node.

6.1 Finding a Failure Node

First, the coloring algorithm is adapted to identify and remember failure nodes. In addition, for each node n that is colored by ?, but is *not* a failure node, the coloring algorithm remembers a son that was already colored ? by the time n was colored, denoted $cont(n)$. This is a node to which the search for a failure node should continue from n, hence the name. Now, given the initial node n_0 that is colored ? a failure node is found by the algorithm `FailureSearch` shown in Figure 6, starting from n_0. This is a DFS-like greedy algorithm, that proceeds (recursively) from node to node, using $cont(n)$, until it encounters a failure node.

LEMMA 6.2. *The algorithm* `FailureSearch` *terminates.*

LEMMA 6.3. *A failure node, and in particular the one returned by the algorithm* `FailureSearch`*, is a node colored by ?, which is one of the following.*

(1) A terminal node of the form (s_a, l) where $l \in Lit$.

(2) An AX-node (EX-node) that has a may-son colored by F (T).

(3) An AX-node (EX-node) that was colored during phase 2a based on an AU (EV) witness, and has a may-son colored by ?.

6.2 Failure Analysis

Given a failure node $n = (s_a, \varphi)$, it provides us with criteria for the refinement. This is done in two steps. First, we determine how the set of concrete states represented by s_a should be split in order to discard the indefinite color of n. We then decide how to alter the abstraction in order to ensure the required split of s_a. To solve the latter problem, a criterion for splitting *all* abstract states, while ensuring the split of s_a, can be found by known techniques, depending on the type of abstraction used. Examples of solutions for certain types of abstractions can be found in [Clarke et al. 2002] (for abstraction based on invisible variables) and in [Clarke et al. 2000] (for abstraction based on formulae clusters). It remains to explain which subsets of $\gamma(s_a)$ need to be separated by the refinement. The criterion for the separation depends on the type of n and is found by the following analysis.

(1) $n = (s_a, l)$ is a terminal node. In this case, its indefinite color results from the fact that s_a represents concrete states that are labeled by l as well as concrete

states that are labeled by $\neg l$. In order to avoid the indefinite color in this node, we need to separate these types of concrete states. Hence, our goal is to separate $\gamma(s_a)$ to two sets $\{s_c \in \gamma(s_c) : l \in L_C(s_c)\}$ and $\{s_c \in \gamma(s_c) : \neg l \in L_C(s_c)\}$.

(2) $n = (s_a, AX\varphi_1)$ with a may-son colored F, or $n = (s_a, EX\varphi_1)$ with a may-son colored T. Let K stand for F or T. We define $sons_K = \bigcup\{\gamma(s'_a) : (s'_a, \varphi_1)$ is a may-son of n colored $K\}$ and $conc_K = \gamma(s_a) \cap \{s_c \in S_C : \exists s'_c \in sons_K, s_c \rightarrow s'_c\}$. For the $AX\varphi_1$ case, $K = F$ and $conc_K$ is the set of all concrete states, represented by s_a, that definitely refute $AX\varphi_1$. For the $EX\varphi_1$ case, $K = T$ and $conc_K$ is the set of all concrete states, represented by s_a, that definitely satisfy $EX\varphi_1$. In both cases, our goal is to separate the sets $conc_K$ and $\gamma(s_a) \setminus conc_K$.

(3) $n = (s_a, AX\varphi_1)$ or $n = (s_a, EX\varphi_1)$ was colored during phase 2a based on an AU or an EV witness (respectively), and has a may-son $n' = (s'_a, \varphi_1)$ colored by ?. Let $conc_? = \gamma(s_a) \cap \{s_c \in S_C : \exists s'_c \in \gamma(s'_a), s_c \rightarrow s'_c\}$ be the set of all concrete states, represented by s_a, that have a son represented by s'_a. Our goal is to separate the sets $conc_?$ and $\gamma(s_a) \setminus conc_?$.

It is possible that one of the sets obtained during the failure analysis is empty and provides no criterion for the split. Yet, new information can be gained from it as well. For example, consider case 2, where the failure node n is an AX-node. If $conc_F = \gamma(s_a)$, then in fact every state that is represented by s_a has a refuting son. Thus, n can be colored by F instead of its current indefinite color ?. If $conc_F = \emptyset$, then in fact none of the concrete states in $\gamma(s_a)$ has a transition to a concrete state that is represented by the F-colored may-sons of n. Thus, the (abstract) may-edges from n to such sons can be removed. Either way, the game-graph can be recolored based on the new information, starting from the may-MSCC containing n. Similar arguments apply to the rest of the cases as well.

The intuition behind the criterion for the split that is derived from cases 1-2 is clear. Its purpose is to allow us to conclude definite results about (at least part) of the new abstract states obtained by the split of the failure node. We now consider case 3. In this case we know that by the time the failure node n got colored, its may-son n' that is colored by ? was not yet colored (otherwise n would not be a failure node). By the description of the coloring algorithm, we know that if n' was a must-son of n, then as long as it was uncolored, n would remain uncolored too and would eventually be colored in phase 2b by a definite color. Thus, our goal in this case is on the one hand to obtain a must edge between (parts of) n and n' and on the other hand remove the may-edge altogether between other parts of n and n'.

Example 6.4. Consider the game-graph described in Figure 7(a). Its initial node is colored by ?, thus refinement is needed. To understand which node is a failure node and to identify $cont(n)$ for a node n that is colored by ? but is not a failure node, we need to follow the coloring algorithm. All the nodes in $Q_1 - Q_4$ are terminal nodes and are colored accordingly by definite colors. As for Q_5, the nodes n_6, n_5, and n_4 are colored in phase 1 by definite colors. The rest are colored in phase 2a by ? as follows. The nodes n_3, n_2, n_1 and $n_0 = (s, A(pUq))$ are colored in this order, which makes n_3 a failure node, whereas n_2, n_1 and n_0 are not failure nodes and for

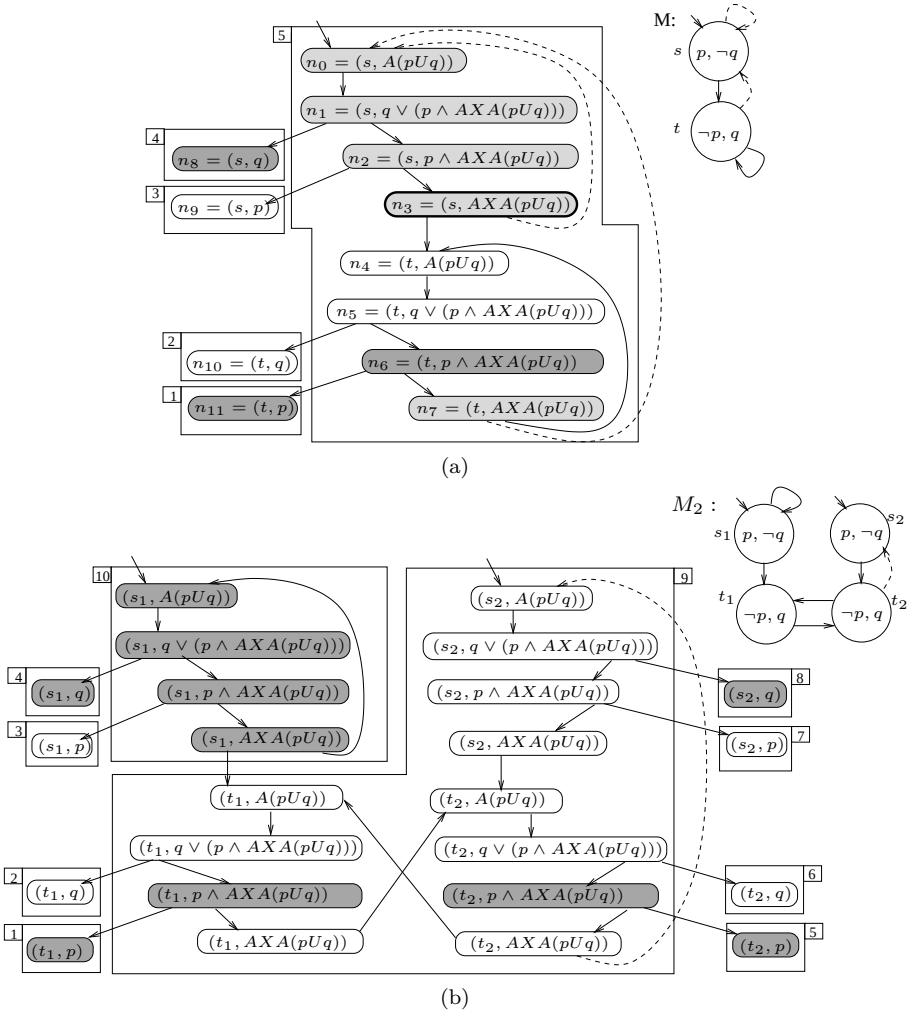

Fig. 7. (a) A 3-valued colored game-graph for M and $\varphi = A(pUq)$, where the initial node is colored by ? and the failure node found by `FailureSearch` appears in boldface; (b) The colored refined game-graph, based on the refined model M_2 and φ.

them $cont(n_0) = n_1$, $cont(n_1) = n_2$ and $cont(n_2) = n_3$. The node n_7 can be colored at any time during this phase. If it is colored before n_0, then it is also a failure node, otherwise $cont(n_7) = n_0$. Given this information, the algorithm `FailureSearch` is applied starting from the initial node, $n_0 = (s, A(pUq))$. It continues to n_1, from there to n_2 and then reaches the failure node $n_3 = (s, AXA(pUq))$.

Given the failure node n_3, failure analysis is applied. n_3 is an AX-node. It does not have a may-son colored by F, thus as guaranteed by Lemma 6.3, it has a may-son n_0 that is colored by ? and it was colored in phase 2a based on an AU-witness. That is, n_3 corresponds to the third case. Thus, we compute the set $conc_? = \gamma(s) \cap \{s_c \in S_C : \exists s'_c \in \gamma(s), s_c \rightarrow s'_c\}$, which is the set of all concrete

states represented by s that have a transition to a concrete state represented by s. The refinement is aimed at separating the sets $conc_?$ and $\gamma(s) \setminus conc_?$. For example, if the abstraction that is used is based on making some of the variables of the system *invisible*, then these sets may be separated by turning some of the invisible variables *visible* again.

Figure 7(b) presents a possible refined model M_2, where each abstract state was split into two abstract states (possibly due to the addition of one visible variable) and the abstract transitions were computed according to the new abstract states. It also presents the resulting refined game-graph, where the split in the abstract states caused the nodes of the game-graph to be split as well. For example, n_0 was split to $(s_1, A(pUq))$ and $(s_2, A(pUq))$. n_3 was split to $(s_1, AXA(pUq))$ and $(s_2, AXA(pUq))$, etc. It can be seen that in the refined game-graph the initial nodes are now colored by definite colors.

In this example the may-edge from the failure node n_3 to n_0 that existed in the game-graph described in Figure 7(a) and guided the refinement was indeed eliminated: it no longer exists as a may-edge in the refined game-graph described in Figure 7(b) between none of the nodes that resulted from n_3 and n_0. The nodes $(s_1, AXA(pUq))$ and $(s_1, A(pUq))$ that resulted from them now have a *must-edge* between them, whereas the node $(s_2, AXA(pUq))$, which also resulted from n_3 does not have *any* edge to the nodes $(s_1, A(pUq))$ and $(s_2, A(pUq))$ that resulted from n_0.

Note that in failure nodes of the type AX or EX we have ignored the information about sons that are colored by T or F respectively. This information may be used to derive criteria for further separation of $\gamma(s_a)$. For example, consider the case of an AX-node. Let $\widetilde{conc}_T = \gamma(s_a) \cap \overline{\{s_c \in S_C : \exists s'_c \in \overline{sons_T}, s_c \to s'_c\}}$ (where $\overline{A} \equiv S_C \setminus A$) be the set of all the concrete states represented by s_a, that *all* their (concrete) sons are represented by sons of n that are colored by T. These states all satisfy the AX formula. Thus, separating them from the rest may be helpful as well.

Having suggested a refinement mechanism that suits the 3-valued model checking algorithm of Section 4, we now have all the components required for an iterative abstraction-refinement framework, where each iteration consists of abstraction, model checking and refinement.

THEOREM 6.5. *For finite concrete models, iterating the abstraction-refinement process is guaranteed to terminate with a definite answer.*

7. INCREMENTAL ABSTRACTION-REFINEMENT FRAMEWORK

We refine abstract models by splitting their states. The criterion for the refinement is decided locally, based on one node, but it has a global effect, since the refinement is applied to the whole abstract model. Yet, in practice, there is no reason to split states for which the model checking results are definite. The game-based model checking algorithm provides us with a convenient framework to use previous results and avoid unnecessary refinement. This leads to an *incremental* model checking algorithm based on iterative abstraction-refinement. This section is devoted to the description of our incremental abstraction-refinement framework.

At the end of the ith iteration of abstraction-refinement, we now remember the

(abstract) nodes that were colored by definite colors, as well as nodes for which a definite color was discovered during the failure analysis. Let D_i denote the set of such nodes. Let $\chi_{D_i} : D_i \to \{T, F\}$ be the coloring function that maps each node in D_i to its (definite) color. χ_{D_i} can be extracted from the result of the coloring algorithm, along with the failure analysis phase. In order to describe how the incremental approach uses this information we need the following definition.

Definition 7.1. Let M_C be a concrete model, and let M_A, M'_A be two abstract KMTSs of M_C, with concretization functions γ and γ' respectively. Let φ be a CTL formula, and let G_A and G'_A be the game-graphs, constructed based on φ and the models M_A and M'_A respectively. We say that a node $n_a = (s_a, \varphi_1)$ of G_A is a *sub-node* of a node $n'_a = (s'_a, \varphi'_1)$ of G'_A if (1) $\varphi_1 = \varphi'_1$, and (2) $\gamma(s_a) \subseteq \gamma'(s'_a)$.

That is, a node (s_a, φ_1) is a *sub-node* of (s'_a, φ'_1) if they both have the same subformula, and the set of concrete states represented by s_a is a subset of those represented by s'_a.

Consider the jth iteration of abstraction-refinement. Let $D = \bigcup_{i<j} D_i$ denote the set of nodes that were remembered from previous runs at the jth iteration. These are nodes with definite colors. Let $\chi_D : D \to \{T, F\}$ denote their coloring function, $\chi_D = \bigcup_{i<j} \chi_{D_i}$. The incremental approach uses this information both in the construction of the game-graph and in its coloring.

Game-graph construction: During the construction of a new refined game-graph in the jth iteration, we prune the game-graph in nodes that are *sub-nodes* of nodes from D. When a node n that is a sub-node of a node $n_d \in D$ is encountered, we add $n_d \in D$ to the game-graph rather than n and do not continue to construct the game-graph from n, nor n_d. As a result of this pruning, only the reachable subgraph that was previously colored by ? is refined.

Coloring: The coloring algorithm then considers the nodes in D, where the game-graph was pruned, as terminal nodes and colors them by their previous colors, i.e. a node $n_d \in D$ is colored by $\chi_D(n_d)$. The rest of the algorithm remains unchanged. This is similar to the partial coloring algorithm, presented in Definition 3.1.

Note that for many abstractions, checking if a node is a sub-node of another is simple. For example, in the framework of predicate abstraction [Graf and Saidi 1997; Saidi and Shankar 1999; Namjoshi and Kurshan 2000; Godefroid et al. 2001], this means that the abstract states "agree" on all the predicates that exist before the refinement. When the abstraction is based on invisible variables [Clarke et al. 2002], this means that the abstract states "agree" on all the variables that are visible before the refinement.

The correctness of the incremental approach is given by the following theorem.

THEOREM 7.2. *Let M_C be a concrete model, and let M_A be an abstract KMTS, such that $M_C \preceq M_A$, with a concretization function γ. Let G_A be the game-graph for M_A and some CTL formula φ, constructed and colored by the incremental algorithm. Then, for each $n_a = (s_a, \varphi_1) \in G_A$:*

(1) If n_a is colored T, then $\forall s_c \in \gamma(s_a) : [(M_C, s_c) \models \varphi_1] = tt$.
(2) If n_a is colored F, then $\forall s_c \in \gamma(s_a) : [(M_C, s_c) \models \varphi_1] = ff$.

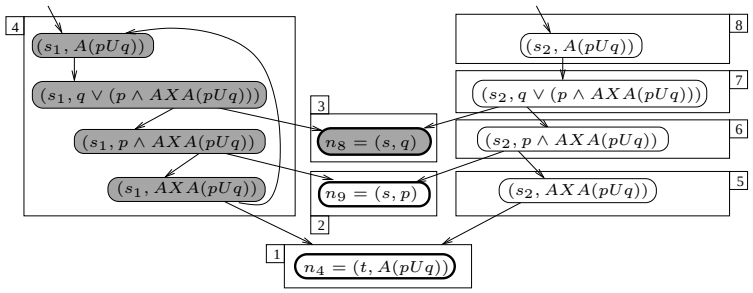

Fig. 8. The pruned game-graph for M_2 and φ from Figure 7, where nodes from the initial game-graph, presented in Figure 7(a), appear in boldface.

Note, that Theorem 7.2 refers to the value of φ_1 in the concrete states represented by s_a and not in s_a itself. The value in s_a, based on the 3-valued semantics, may be indefinite. This means that Theorem 4.7 that relates the coloring and the 3-valued semantics no longer holds.

Example 7.3. Figure 8 demonstrates the use of previous results within the incremental algorithm, where after refining the model M described in Figure 7(a), instead of building the entire game-graph based on the refined model M_2, as described in Figure 7(b), the game-graph is pruned in $(t_1, A(pUq))$ and $(t_2, A(pUq))$ that are both sub-nodes of n_4 which already had a definite color (T). The same goes for (s_1, q) and (s_2, q) that are sub-nodes of n_8, and for (s_1, p) and (s_2, p) that are sub-nodes of n_9. The pruned game-graph, presented in Figure 8, is clearly smaller and simpler than the full refined game-graph. Its coloring handles the nodes in $Q_1 - Q_3$, where the game-graph was pruned, as terminal nodes and colors them by their previous colors. The nodes in Q_4 are all colored by F in phase 2b, based on the AU-witness, whereas the nodes in $Q_5 - Q_8$ are colored in phase 1.

8. CONCLUSION

In this work, we have exploited the game-theoretic approach of CTL model checking to produce annotated counterexamples for full CTL. We have generalized this approach to 3-valued abstract models and suggested an incremental abstraction-refinement framework based on our generalization.

Traditional game-based model checking algorithms determine a winning strategy for the winning player. The winning strategy holds all the relevant information as for the result of the model checking, but it has redundancies. The annotated counterexample introduced in this paper may be seen as a minimal part of it that is sufficient to explain the result. It may be improved and optimized in several ways.

Our 3-valued game-based model checking and in particular the failure nodes provide information for refinement, in case the outcome is indefinite. Additional information can be extracted from them and be used for further optimizations of the refinement.

The incremental abstraction-refinement algorithm described in this paper can be viewed as a generalization of Lazy abstraction [Henzinger et al. 2002], which allows

different parts of the abstract model to exhibit different degrees of abstraction. Lazy abstraction refers to safety properties, whereas our approach is applicable to full CTL.

This work is based on the game-theoretic approach to model checking. This approach is closely related to the Automata-theoretic approach [Kupferman et al. 2000], as described in [Leucker 1999]. Thus, our work can also be described in this framework, using alternating automata. In addition, it can easily be extended to alternation-free μ-calculus.

APPENDIX

A. DISCUSSION: 2-VALUED GAME-BASED MODEL CHECKING

In our discussion on abstract models, we have used the 3-valued semantics for the interpretation of a CTL formula over a KMTS. The definition of $[(M,s) \models \varphi]$ can be extended to a KMTS using a *2-valued semantics* as well [Dams et al. 1997], where the possible truth values are tt and ff. The definition is similar to the concrete semantics with the following changes. Universal properties, of the form $A\psi$, are interpreted along *may* paths. Existential properties, of the form $E\psi$, are interpreted along *must* paths. This gives us the *2-valued semantics* of CTL formulae over KMTSs, denoted $[(M,s) \stackrel{2}{\models} \varphi] = \mathrm{tt} \;/\; = \mathrm{ff}$. The 2-valued semantics is designed to preserve the truth of a formula from the abstract model to the concrete one. However, false alarms are possible, where the abstract model falsifies the property, but the concrete one does not.

THEOREM A.1. *[Dams et al. 1997] Let $H \subseteq S_C \times S_A$ be a mixed simulation relation from a Kripke structure M_C to a KMTS M_A. Then for every $(s_c, s_a) \in H$ and every CTL formula φ, we have that: $[(M_A, s_a) \stackrel{2}{\models} \varphi] = tt$ implies that $[(M_C, s_c) \stackrel{2}{\models} \varphi] = tt$. We conclude that if $M_C \preceq M_A$, then $[M_A \stackrel{2}{\models} \varphi] = tt$ implies that $[M_C \stackrel{2}{\models} \varphi] = tt$.*

The game-based model checking algorithm can be extended to deal with KMTSs based on the 2-valued semantics in a more natural way than was needed to deal with the 3-valued semantics.

The 2-valued semantics is aimed at proving φ: it preserves only truth from the abstract model to the concrete one. Therefore, the purpose of the game is also to prove φ's satisfaction. As such, the moves of $\exists$loise in configurations with $EX\varphi'$ formulae need to use $\stackrel{\text{must}}{\longrightarrow}$ transitions, since by the semantics definition, existential formulae are interpreted over *must*-paths. Similarly, the moves of $\forall$belard in configurations with $AX\varphi'$ formulae need to use $\stackrel{\text{may}}{\longrightarrow}$ transitions, since universal formulae are interpreted over *may*-paths. The rest of the moves, as well as the winning criteria remain the same, with the following exception. The transition relation $\stackrel{\text{must}}{\longrightarrow}$ is not necessarily total. Thus, a configuration of the form $(s, EX\varphi')$ may also be a terminal configuration, if s has no outgoing $\stackrel{\text{must}}{\longrightarrow}$ transitions. A play that ends in such a configuration is won by $\forall$belard.

Clearly, the relation between the existence of winning strategies and the satisfaction of the formula, as described in Theorem 2.5 for the concrete game, holds for the new game and the 2-valued semantics. This results from the fact that the

change in the allowed moves of the players corresponds exactly to the change in the 2-valued interpretation of a formula over a KMTS.

The model checking algorithm, induced by the game consists of two parts: construction of a game-graph based on the rules of the game, and its coloring. Once the moves for the new game are defined, the game-graph is defined as well. Recall that in the 3-valued case, the resulting game-graph had a different structure and thus the coloring algorithm needed to be changed as well. However, in the 2-valued case, the resulting game-graph has the same structure as a concrete game-graph (with the exception of a new type of terminal nodes): Although the abstract model has two types of transitions for each state, when the game-graph is constructed, the edges become uniform. We no longer distinguish between them, since there is only one type in each node. As a result, in terms of the game-graph there is only one type of edges. Thus, the same coloring algorithm can be applied on the (abstract) game-graph in order to check which player has a winning strategy, with the small change that terminal nodes of the form $(s, EX\varphi')$ need to be colored by F. The correctness of the coloring algorithm, as described in Theorem 2.8 for the concrete case, is maintained since the new game has the same properties as a concrete game: the same possible moves from each configuration (with the type of transitions adapted to match the semantics) and the same winning rules. Thus we are guaranteed that the game-graph is colored by the color of the player that has a winning strategy.

Altogether, we get that the resulting coloring function corresponds to the truth value of the formula over the abstract model, under the 2-valued interpretation of a formula over a KMTS. This is formalized by the following theorem.

THEOREM A.2. *Let M be a KMTS and φ a CTL formula. Then, for each $n = (s, \varphi_1) \in G_{M\times\varphi}$:*

(1) $[(M, s) \stackrel{2}{\models} \varphi_1] = tt$ *iff* $n = (s, \varphi_1)$ *is colored by* T.

(2) $[(M, s) \stackrel{2}{\models} \varphi_1] = ff$ *iff* $n = (s, \varphi_1)$ *is colored by* F.

Intuitively speaking, a node marked by (s, $AX\varphi'$) is now colored by T iff all its sons in the game-graph are colored by T (satisfy φ'), where its sons represent all the states to which s has may transitions. In a similar way, a node marked by (s, $EX\varphi'$) is colored by T iff one of its sons is colored by T (satisfies φ'), where this son represents a state to which s has a must transition. Therefore, the coloring corresponds to the 2-valued semantics.

Complexity. Clearly, the running time of the coloring algorithm remains linear with respect to the size of the game-graph $G_{M\times\varphi}$. The latter is bounded by the size of the underlying KMTS times the length of the CTL formula, i.e. $O(|M| \cdot |\varphi|)$.

A.1 Application to 3-Valued Model Checking

We have the following correspondence between the 2-valued semantics and the 3-valued semantics.

THEOREM A.3. *Let M be a KMTS. Then for every CTL formula φ and for every $s \in S$, we have that:*

(1) if $[(M,s) \overset{2}{\models} \varphi] = tt$ *then* $[(M,s) \overset{3}{\models} \varphi] = tt$.

(2) if $[(M,s) \overset{2}{\models} \neg\varphi] = tt$ *then* $[(M,s) \overset{3}{\models} \varphi] = ff$.

(3) otherwise $[(M,s) \overset{3}{\models} \varphi] = \bot$.

where $\neg\varphi$ denotes the CTL formula equivalent to $\neg\varphi$, with negations pushed to the literals.

Based on Theorem A.3, given an abstract KMTS M_A such that $M_C \preceq M_A$, one may suggest using two instances of the previously described 2-valued model checking algorithm in order to evaluate the 3-valued truth value of φ over M_A, as follows.

First, evaluate φ over M_A using the 2-valued semantics. The constructed game-graph is referred to as the *satisfaction* graph, since it was built for the purpose of proving the satisfaction of φ. If the result is tt for all the initial states, then we have that $[M_A \overset{3}{\models} \varphi] = \text{tt}$ and we can conclude that $[M_C \models \varphi] = \text{tt}$.

Otherwise, evaluate $\neg\varphi$ (with negations pushed to the literals) over M_A using the 2-valued semantics. The constructed game-graph is referred to as the *refutation* graph, since it was built for the purpose of proving satisfaction of the negation of φ (which is equivalent to proving refutation of φ). If the result is tt for at least one initial state, we have that $[M_A \overset{3}{\models} \varphi] = \text{ff}$ and we can conclude that $[M_C \models \varphi] = \text{ff}$. In addition, a concrete annotated counterexample may be produced from the refutation graph.

This can be better understood using the following observation. Note, that instead of evaluating $\neg\varphi$ using the previous 2-valued game-based model checking algorithm, it is possible to define a game with different rules that is designed to refute the formula φ. In such a game the players use the opposite type of transitions in each configuration (node): $\forall$belard uses must transitions in AX-nodes and $\exists$loise uses may-transitions in EX-nodes. As a result, F is preserved from the corresponding abstract game-graph to the concrete one, but T is not. Note, that the game-graph obtained by these rules is isomorphic to the refutation graph and the result of its coloring is equivalent to the result of the previous algorithm applied on $\neg\varphi$. Obviously, if an initial node in such a game-graph is colored by F, then we can easily find an abstract annotated counterexample by the algorithm `ComputeCounter` described in Section 3. The abstract annotated counterexample is guaranteed not to be spurious and can be matched with a concrete one by a greedy algorithm, as described in Section 5.

If none of the above holds, we have that $[M_A \overset{3}{\models} \varphi] = \bot$. Thus, M_A needs to be refined. One would suggest to try and use both the satisfaction graph and the refutation graph and their coloring functions to find a criterion for refinement. In a sense they complement each other, because they are based on opposite types of transitions. However, these two game-graphs have different nodes (because reachability is also based on opposite transitions), so most chances are that we can not find enough needed information in their intersection. This is demonstrated in Figure 9, where in the satisfaction graph (a) the initial node $(s, AX\varphi)$ is colored by F

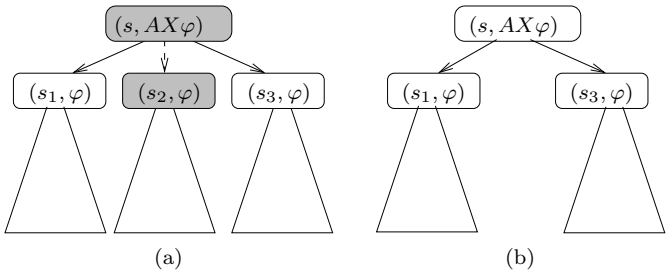

Fig. 9. A satisfaction graph (a) versus a refutation graph (b) for $AX\varphi$

since its son (s_2, φ) is colored by F. However, in the refutation graph (b) $(s, AX\varphi)$ is colored by T. Thus, the result of the model checking in indefinite. Unfortunately, the refuting son from the satisfaction graph, (s_2, φ), and the whole subgraph originated from it, do not appear in the refutation graph, since the refuting son is not a must-son of $(s, AX\varphi)$. Thus, combining the information of both these graphs does not supply enough information for the refinement.

In summary, this approach provides the same information as the 3-valued algorithm about nodes that appear in both the satisfaction and the refutation graphs. Since the *initial* nodes appear in both of them, this approach is sufficient in order to answer the question "what is the value of $[M \overset{3}{\models} \varphi]$?", as accurately as the direct 3-valued approach. However, for the refinement analysis we are interested in the *inner* nodes as well, that are not necessarily mutual to both graphs. Thus, using two such game-graphs does not provide us with full information (in terms of edges) about all of them. Hence, the 3-valued game-based algorithm is advantageous in terms of the refinement.

Note, that this approach is similar in spirit to the result of translating the KMTS to an equivalent partial Kripke structure (PKS) as described in [Godefroid and Jagadeesan 2003] and then model checking the PKS under the 3-valued semantics by running a standard 2-valued model checker twice, as described in [Bruns and Godefroid 2000].

B. MEMORYLESS WINNING STRATEGIES

Theorem 4.2 describes the correspondence between the existence of winning strategies for the model checking game and the model checking results. Thus, it refers to the existence (or non-existence) of winning strategies for the players. Recall that a strategy for a player is a set of rules telling him (or her) how to proceed from a given configuration. In general, the rules in the strategy can depend on the entire sequence of configurations in the current play that led to the given configuration. Yet, one may also be interested in referring to *memoryless* (or *history-free*) winning strategies, where every rule can depend only on the current configuration of the play and cannot depend on the course of the play up to this configuration (i.e., it should be independent of the prefix of the play). More formally:

Definition B.1. A *strategy σ for a player P* is a function assigning to every finite sequence of configurations $\vec{C}$ ending in a configuration C, which is the responsibility

of the player P, a configuration C', such that the move from C to C' is a legal move for P. A strategy is *memoryless* iff $\sigma(\vec{C}) = \sigma(\vec{C'})$ whenever $\vec{C}$ and $\vec{C'}$ end in the same configuration.

The winning strategies described in the proof of Theorem 4.2 are not necessarily memoryless. The reason for this is that the proof constructs winning strategies based on paths of the model and the paths that are used for this purpose may contain repetitions of states. The result is that it is possible that the strategy contains different rules for the same configuration (s', φ') based on the position of the state s' in the path, or in other words, based on the position of the configuration in the play. This problem is avoided when using *simple* paths.

Definition B.2. —An *infinite* path π is said to be *simple* if π is of the form $x \cdot y^\omega$, where $x = s_0, \ldots, s_k$ and $y = s_{k+1}, \ldots, s_n$ such that for every $0 \leq i, j \leq n$: $i \neq j \implies s_i \neq s_j$.

—A *finite* path π is said to be *simple* if π is of the form $s_0, \ldots, s_n$ such that for every $0 \leq i, j \leq n$: $i \neq j \implies s_i \neq s_j$.

The following lemma implies that every non-simple path that is used in the proof of Theorem 4.2 for the construction of a strategy for any of the players can be replaced by a simple one.

LEMMA B.3. *Let ψ be a (path) formula of the form $\varphi_1 U \varphi_2$ or $\varphi_1 V \varphi_2$, where φ_1 and φ_2 are CTL formulae. If there exists a (must or may) path π such that $[\pi \models^3 \psi] = val$ for $val \in \{tt, ff, \bot\}$, then there exists a* simple *path π' of the same type (must or may) such that $[\pi' \models^3 \psi] = val$ as well.*

Based on Lemma B.3, we conclude that the winning strategies in the proof of Theorem 4.2 can be made memoryless. As a result we get that Theorem 4.2 can be rephrased in terms of *memoryless* winning strategies.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tocl/2006-V-N/p1-URLend`.

REFERENCES

ASTEROTH, A., BAIER, C., AND ASSMANN, U. 2001. Model checking with formula-dependent abstract models. In Computer-Aided Verification (CAV). *Lecture Notes in Computer Science 2102*, 155–168.

BARNER, S., GEIST, D., AND GRINGAUZE, A. 2002. Symbolic localization reduction with reconstruction layering and backtracking. In *Proc. of Conference on Computer-Aided Verification (CAV)*. Copenhagen, Denmark.

BOLLIG, B., LEUCKER, M., AND WEBER, M. 2002. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*. Springer-Verlag Inc.

BRUNS, G. AND GODEFROID, P. 1999. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*. 274–287.

BRUNS, G. AND GODEFROID, P. 2000. Generalized model checking: Reasoning about partial state spaces. In CONCUR'00. *Lecture Notes in Computer Science 1877*, 168–182.

Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., and D.Wang. 2002. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD)*.

Chechik, M., Devereux, B., and Easterbrook, S. 2001. Implementing a multi-valued symbolic model checker. In *TACAS'01*. 404–419.

Chechik, M., Gurfinkel, A., and Devereux, B. 2002. chi-chek: A multi-valued model-checker. In *CAV'02*. 505–509.

Clarke, E. and Emerson, E. 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logics of Programs*. Volume 131, LNCS, Springer-Verlag, Berlin, 52–71.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2000. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV'00)*. LNCS. Chicago, USA.

Clarke, E., Grumberg, O., McMillan, K., and Zhao, X. 1995. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd Design Automation Conference (DAC'95)*. IEEE Computer Society Press.

Clarke, E., Grumberg, O., and Peled, D. 1999. *Model Checking*. MIT press.

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

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

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *popl4*. Los Angeles, California, 238–252.

Dams, D., Gerth, R., and Grumberg, O. 1997. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19,* 2 (March).

Das, S., Dill, D., and Park, S. 1999. Experience with predicate abstraction. In *Computer Aided Verification*. 160–171.

Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., and Campenhout, D. V. 2003. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*. LNCS, vol. 2725. Springer, Boulder, CO, USA, 27–39.

Emerson, E. and Clarke, E. 1982. Using branching time logic to synthesize synchronization skeletons. *Sci. Comput. Program. 2*, 241–266.

Godefroid, P., Huth, M., and Jagadeesan, R. 2001. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR'01*.

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

Godefroid, P. and Jagadeesan, R. 2003. On the expressiveness of 3-valued models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*. LNCS, vol. 2575. Springer-Verlag, New York, 206–222.

Govindaraju, S. and Dill, D. 1998. Verification by approximate forward and backward reachability. In *ICCAD*. 366–370.

Graf, S. and Saidi, H. 1997. Construction of abstract state graphs with PVS. In *Proc. of Conference on Computer-Aided Verification (CAV)*. LNCS, vol. 1254. Springer-Verlag, 72–83.

Gurfinkel, A. and Chechik, M. 2003a. Generating counterexamples for multi-valued model-checking. In *FME'03*. 503–521.

Gurfinkel, A. and Chechik, M. 2003b. Proof-like counter-examples. In *Proceedings of TACAS'03*.

Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM Press, Portland, Oregon, 58–70.

Huth, M., Jagadeesan, R., and Schmidt, D. 2001. Modal transition systems: A foundation for three-valued program analysis. In European Symposium on Programming (ESOP'01). *LNCS 2028*, 155–169.

Kupferman, O., Vardi, M., and Wolper, P. 2000. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM) 47*, 2, 312–360.

Kurshan, R. 1994. *Computer-Aided-Verification of Coordinating Processes*. Princeton University Press.

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

Larsen, K. G. 1989. Modal specifications. In *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, J. Sifakis, Ed. Lecture Notes in Computer Science, vol. 407. Springer-Verlag.

Lee, W., Pardo, A., Jang, J., Hachtel, G., and Somenzi, F. 1996. Tearing based automatic abstraction for CTL model checking. In *ICCAD*. 76–81.

Leucker, M. 1999. Model checking games for the alternation free mu-calculus and alternating automata. In *6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*.

Lichtenstein, O. and Pnueli, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. 97–107.

Lind-Nielsen, J. and Andersen, H. 1999. Stepwise CTL model checking of state/event systems. In *Computer Aided Verification*. 316–327.

Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. 1995. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design 6*, 11–45.

Namjoshi, K. 2001. Certifying model checkers. In *13th Conference on Computer Aided Verification (CAV)*. LNCS, vol. 2102. Springer-Verlag.

Namjoshi, K. and Kurshan, R. 2000. Syntactic program transformations for automatic abstraction. In *Proc. of Conference on Computer-Aided Verification (CAV)*. LNCS, vol. 1855. Springer, Chicago, IL, USA, 435–449.

Pardo, A. and Hachtel, G. 1997. Automatic abstraction techniques for propositional mu-calculus model checking. In *Computer Aided Verification*. 12–23.

Pardo, A. and Hachtel, G. 1998. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*. 457–462.

Quielle, J. and Sifakis, J. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS, vol. 137. Springer Verlag, 337–351.

Saidi, H. 2000. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS2000)*. Santa Barbara, CA.

Saidi, H. and Shankar, N. 1999. Abstract and model check while you prove. In *Proceedings of the eleventh International Conference on Computer-Aided Verification (CAV99)*. Trento, Italy.

Shoham, S. 2003. A game-based framework for CTL counterexamples and abstraction-refinement. M.S. thesis, Department of Computer Science, Technion - Israel Institute of Technology.

Stirling, C. 2001. *Modal and Temporal Properties of Processes*. Springer.

Tan, L. and Cleaveland, R. 2002. Evidence-based model checking. In *14th Conference on Computer Aided Verification (CAV)*. LNCS, vol. 2404. Springer Verlag, Copenhagen, Denmark, 455–470.

Received October 2003; revised November 2004; accepted November 2005