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

Sharon Shoham and Orna Grumberg

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

Abstract

This work exploits and extends the game-based framework of CTL model checking for counter-example 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 counter-examples*, which are sufficient and minimal counter-examples for full CTL. We present an algorithm that uses the game board of the model checking game to derive an *annotated counter-example* in case the examined system model refutes the checked formula.

1 Introduction

This work exploits and extends the game-based framework [33] of CTL model checking for counterexample and incremental abstraction-refinement.

The first goal of this work is to suggest a game-based new model checking algorithm for the branching-time temporal logic CTL [8] in the context of abstraction. Model checking 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. However, if the abstract model does not satisfy the formula then nothing is known about the concrete model.

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 [16] 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.

Following the above observation, we define a game-based model checking algorithm for abstract models w.r.t. the 3-valued semantics, where the abstract model can be used for both verification and falsification. However, a third case is now possible: model checking may end with an indefinite answer. This is an

indication that our abstraction cannot determine the value of the checked property in the concrete model and therefore needs to be refined. The traditional abstraction-refinement framework [21, 7] is designed for 2-valued abstractions, where false may be a false-alarm, thus refinement is aimed at eliminating false results. As such, it is usually based on a counterexample analysis. Unlike this approach, the goal of our refinement is to eliminate indefinite results and turn them into either definite true or definite false.

An advantage of this work lies in the fact that the refinement is then 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 use the game-based framework in order 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]) [6, 8]. Recently, this approach has been extended to provide counterexamples for all formulae of the universal branching-time temporal logic ACTL [10]. In this case the part of the model given as the counterexample has the form of a tree. Other works also extract information from model checking [31, 13, 27, 34]. However, this information is presented in the form of a temporal proof, rather than a part of the model.

In this work we provide counterexamples for full CTL. As for ACTL, counterexamples are part of the model. However, when 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. We therefore *annotate* each state on the counterexample 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 propose an algorithm that constructs an annotated counterexample and prove that it is sufficient and minimal. We also discuss several ways to use and present this information in practice.

Games for CTL model checking [33] is 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. $\forall$belard proceeds from such a node (s, ψ) to a node that helps refuting ψ on s. $\exists$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 $\forall$belard has a winning strategy (i.e., it can win the game regardless of $\exists$loise moves) iff $M \not\models \varphi$. $\exists$loise has a winning strategy iff $M \models \varphi$.

Model checking is then done by applying a *coloring algorithm* on the game-graph [3]. It colors a node (s, ψ) by T iff $\exists$loise has a winning strategy, which means ψ is true in s. It colors it by F iff $\forall$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)

[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".

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 [22, 12] 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. We would like to maintain the property of the 3-valued semantics that both the positive and the negative answers are definite in the sense that they hold for the concrete model as well. 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 $M \models \varphi$, and ∃loise has a winning strategy iff $M \not\models \varphi$. However, it is also possible that none 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. However, the structure of the game-graph allows us to apply it only to the indefinite part of the model. It also allows us to use definite results that were obtained previously. Thus, previous runs are not wasted and the abstract model does not grow where it is not needed.

Other researchers [16] have suggested to evaluate a property w.r.t the 3-valued semantics by reducing the problem to two 2-valued model checking problems: one for satisfaction and one for refutation. Such a reduction will result in the same answer as our algorithm. Yet, it is then not clear how to guide the refinement, in case it is needed, since at least part of the information about the indefinite portion of the game-graph is lost. Thus, the application to refinement demonstrates the advantage of designing a 3-valued model checking algorithm.

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 needs to be chosen wisely. 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.

Related Work. Other researchers have suggested abstraction-refinement mechanisms for various branching time temporal logics. In [23] 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 [29, 30] 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 representation. In [25] 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. [1] 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.

The rest of the paper is organized as follows. In the next section we give the necessary background for game-based CTL model checking, abstractions and the 3-valued semantics. Due to technical reasons, we 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. In Section 5 we present our refinement technique, as well as an incremental abstraction-refinement framework. Finally, we discuss some conclusions in Section 6.

2 Preliminaries

Let AP be a finite set of atomic propositions. We define the set *Lit* of literals over AP to be the set $AP \cup \{\neg p : p \in AP\}$. 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:* $\varphi ::= tt \mid ff \mid l \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\psi \mid E\psi$ *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* $M = (S, S_0, \rightarrow, L)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\rightarrow \subseteq S \times S$ is a transition relation, which must be *total* and $L : S \rightarrow 2^{Lit}$ is a labeling function that associates each state in S with a subset of literals, such that for every state s and atomic proposition $p \in AP$, $p \in L(s)$ iff $\neg p \notin L(s)$. A *path* in M is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that $\forall i \geq 0$, $s_i \rightarrow s_{i+1}$. π is said to be *from* s if $s = s_0$.

$[(M, s) \models \varphi] =$ tt (= ff) means that the CTL formula φ is true (false) in the state s of a Kripke structure M. The formal definition appears in Appendix A. We say that M satisfies φ, denoted $[M \models \varphi] =$ tt, if $\forall s_0 \in S_0 : [(M, s_0) \models \varphi] =$ tt. Otherwise, M refutes φ, denoted $[M \models \varphi] =$ ff. We omit M when clear from the context.

Definition 2.2 *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 *is defined as:*

$$
\begin{array}{lcl}
exp(A(\varphi_1 U \varphi_2)) & = & \{A(\varphi_1 U \varphi_2),\ \varphi_2 \vee (\varphi_1 \wedge AXA(\varphi_1 U \varphi_2)),\ \varphi_1 \wedge AXA(\varphi_1 U \varphi_2),\ AXA(\varphi_1 U \varphi_2)\} \\
exp(E(\varphi_1 U \varphi_2)) & = & \{E(\varphi_1 U \varphi_2),\ \varphi_2 \vee (\varphi_1 \wedge EXE(\varphi_1 U \varphi_2)),\ \varphi_1 \wedge EXE(\varphi_1 U \varphi_2),\ EXE(\varphi_1 U \varphi_2)\} \\
exp(A(\varphi_1 V \varphi_2)) & = & \{A(\varphi_1 V \varphi_2),\ \varphi_2 \wedge (\varphi_1 \vee AXA(\varphi_1 V \varphi_2)),\ \varphi_1 \vee AXA(\varphi_1 V \varphi_2),\ AXA(\varphi_1 V \varphi_2)\} \\
exp(E(\varphi_1 V \varphi_2)) & = & \{E(\varphi_1 V \varphi_2),\ \varphi_2 \wedge (\varphi_1 \vee EXE(\varphi_1 V \varphi_2)),\ \varphi_1 \vee EXE(\varphi_1 V \varphi_2),\ EXE(\varphi_1 V \varphi_2)\}
\end{array}
$$

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 [33, 24]. 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 as usual, except that 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 $sub(\varphi) = exp(\varphi) \cup sub(\varphi_1) \cup sub(\varphi_2)$.

The model checking game is played by two players, $\forall$belard, the refuter, and $\exists$loise, the prover. A single play is a (possibly infinite) sequence $C_0 \rightarrow_{p0} C_1 \rightarrow_{p1} C_2 \rightarrow_{p2} \ldots$ of configurations, where $C_0 \in S_0 \times \{\varphi\}$, $C_i \in S \times sub(\varphi)$ and $p_i \in \{\forall, \exists\}$. The subformula in C_i determines which player p_i makes the next move.

The possible moves at each step are:

1. $C_i = (s, \text{ff})$, $C_i = (s, \text{tt})$, or $C_i = (s, l)$ where $l \in Lit$: the play is finished. Such configurations are called *terminal configurations*.
2. $C_i = (s, AX\varphi)$: ∀belard chooses a transition $s \to s'$ and $C_{i+1} = (s', \varphi)$.
3. $C_i = (s, EX\varphi)$: ∃loise chooses a transition $s \to s'$ and $C_{i+1} = (s', \varphi)$.
4. $C_i = (s, \varphi_1 \wedge \varphi_2)$: ∀belard chooses $j \in \{1, 2\}$ and $C_{i+1} = (s, \varphi_j)$.
5. $C_i = (s, \varphi_1 \vee \varphi_2)$: ∃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 [33] it is shown that a play is infinite iff exactly one $\{AU, EU, AV, EV\}$ subformula occurs in it infinitely often. Such a subformula is called a *witness*.

Winning Criteria: ∀belard wins the play iff (1) the play is finite and ends in a terminal configuration of the form $C_i = (s, \text{ff})$, or $C_i = (s, l)$, where $l \notin L(s)$, or (2) the play is infinite and the witness is AU or EU. ∃loise wins the play otherwise.

The model checking game consists of all the possible plays. A *strategy* is a set of rules for a player, telling him how to move in the current configuration. A *winning strategy* is a set of rules allowing the player to win every play if he plays by the rules.

Theorem 2.3 *[33] 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 for the game starting at* (s, φ).
2. $[(M, s) \models \varphi] = ff$ *iff* *∀belard has a winning strategy for the game 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*, that describes all the possible plays of the game. 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.

2.1.1 Game-graph Construction and its Properties

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 to get the successors in the game-graph of each new node. The result is denoted $G_{M \times \varphi} = (N, E)$, where $N \subseteq S \times sub(\varphi)$. 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 subformulae are of the form $\varphi_1 \wedge \varphi_2$ or $AX\varphi_1$ are ∧*-nodes*.
3. Nodes whose subformulae are of the form $\varphi_1 \vee \varphi_2$ or $EX\varphi_1$ are ∨*-nodes*.

Nodes whose subformulae are AU, EU, AV, EV can be considered either ∨-nodes or ∧-nodes. Sometimes we further distinguish between nodes whose subformulae are 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 to two types. 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*.

Lemma 2.4 *Let B be a non trivial strongly connected component (SCC) in a game-graph (a non-trivial SCC contains at least one edge). Then the set of subformulae that are associated with the nodes in B is exactly one of the sets* $\exp(\varphi)$*, where* $\varphi \in \{A(\varphi_1 U \varphi_2),\ E(\varphi_1 U \varphi_2),\ A(\varphi_1 V \varphi_2),\ E(\varphi_1 V \varphi_2)\}$.

The formula φ such that $exp(\varphi)$ is the set of subformulae in a non-trivial SCC is called a *witness*. Each non-trivial SCC is then classified as an AU, AV, EU, or EV SCC, based on its witness.

2.1.2 Coloring Algorithm

The following *Coloring Algorithm* [3] labels each node 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.

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 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 can be 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 ∃loise wins in them, and by F otherwise.
2. An $\vee$-node ($\wedge$-node) is colored by T (F) if it has a son that is colored by T (F), and by F (T) if all its sons are colored by F (T).
3. 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 2.4 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 2.5 *[33] Let $G_{M\times\varphi}$ be a game-graph and let n be a node in the game-graph, then:*

1. $\chi(n) = T$ *iff* ∃*loise has a winning strategy for the game starting at* n.
2. $\chi(n) = F$ *iff* ∀*belard has a winning strategy for the game starting at* n.

Theorem 2.6 *[33] 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.

2.2 Abstraction

In this section we present abstract models and their relation with concrete models. Abstract models preserving CTL have two transition relations [22, 12]. This is achieved by using *Kripke Modal Transition Systems* [19, 14].

Definition 2.7 *A* Kripke Modal Transition System *(KMTS) is a tuple $M = (S, S_0, \xrightarrow{must}, \xrightarrow{may}, L)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $\xrightarrow{must} \subseteq S \times S$ and $\xrightarrow{may} \subseteq S \times S$ are transition relations such that $\xrightarrow{must} \subseteq \xrightarrow{may}$, 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 an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for every $i \geq 0$, $s_i \xrightarrow{must} s_{i+1}$ ($s_i \xrightarrow{may} s_{i+1}$). If $s = s_0$, then π is said to be* from s.

Note, that a Kripke structure can be viewed as a KMTS where $\rightarrow = \xrightarrow{must} = \xrightarrow{may}$, 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* [26, 12].

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 $(\gamma : S_A \to 2^{S_C}, \alpha : 2^{S_C} \to S_A)$ a *Galois connection* [11, 26] from $(2^{S_C}, \subseteq)$ to $(S_A, \sqsubseteq)$. 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 can then be defined as follows. The set of initial abstract states S_{0A} is defined such that $s_{0a} \in S_{0A}$ iff there exists $s_{0c} \in S_{0C}$ for which $s_{0c} \in \gamma(s_{0a})$. An abstract state s_a is labeled by $l \in Lit$, only if all the concrete states that it represents are labeled by l as well. Thus, 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 they

represent every concrete transition between two states: 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. The *must*-transitions represent concrete transitions that are common to all the concrete states 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 $s_c \rightarrow s'_c$. It is possible that there are less must-transitions than allowed by this rule.

Other constructions of abstract models, based on Galois connections, can be found in [12, 15].

The resulting abstract model is *more abstract* than M_C as defined by the following definition.

Definition 2.8 *[12, 14] 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 $s_a \in S_{0A}$ for which $(s_c, s_a) \in H$, we say that M_A is more abstract *than M_C, denoted by $M_C \preceq M_A$.*

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 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 w.r.t. the mixed simulation relation, and are not limited to our construction of an abstract model.

[19] defines the *3-valued* semantics of a CTL formula over a KMTS, preserving both satisfaction and refutation 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 $\perp$, then its value over the concrete model is not known and can be either tt or ff.

Definition 2.9 *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 $[(M, s) \overset{3}{\models} \varphi]$, is defined inductively as follows:*

$$[(M,s) \overset{3}{\models} \textit{tt}] = \textit{tt}$$

$$[(M,s) \overset{3}{\models} \textit{ff}] = \textit{ff}$$

$$[(M,s) \overset{3}{\models} l] = \begin{cases} \textit{tt} & \textit{if } l \in L(s) \\ \textit{ff} & \textit{if } \neg l \in L(s) \\ \perp & \textit{otherwise} \end{cases}$$

$$[(M,s) \overset{3}{\models} \varphi_1 \wedge \varphi_2] = \begin{cases} \textit{tt} & \textit{if } [(M,s) \overset{3}{\models} \varphi_1] = \textit{tt and } [(M,s) \overset{3}{\models} \varphi_2] = \textit{tt} \\ \textit{ff} & \textit{if } [(M,s) \overset{3}{\models} \varphi_1] = \textit{ff or } [(M,s) \overset{3}{\models} \varphi_2] = \textit{ff} \\ \perp & \textit{otherwise} \end{cases}$$

$$[(M,s) \overset{3}{\models} \varphi_1 \vee \varphi_2] = \begin{cases} \textit{tt} & \textit{if } [(M,s) \overset{3}{\models} \varphi_1] = \textit{tt or } [(M,s) \overset{3}{\models} \varphi_2] = \textit{tt} \\ \textit{ff} & \textit{if } [(M,s) \overset{3}{\models} \varphi_1] = \textit{ff and } [(M,s) \overset{3}{\models} \varphi_2] = \textit{ff} \\ \perp & \textit{otherwise} \end{cases}$$

$$[(M,s) \overset{3}{\models} A\psi] = \begin{cases} \textit{tt} & \textit{if for each may-path } \pi : [(M,\pi) \overset{3}{\models} \psi] = \textit{tt} \\ \textit{ff} & \textit{if there exists a must-path } \pi \textit{ such that } : [(M,\pi) \overset{3}{\models} \psi] = \textit{ff} \\ \perp & \textit{otherwise} \end{cases}$$

$$[(M,s) \overset{3}{\models} E\psi] = \begin{cases} \textit{tt} & \textit{if there exists a must-path } \pi \textit{ such that } : [(M,\pi) \overset{3}{\models} \psi] = \textit{tt} \\ \textit{ff} & \textit{if for each may-path } \pi : [(M,\pi) \overset{3}{\models} \psi] = \textit{ff} \\ \perp & \textit{otherwise} \end{cases}$$

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

$$[(M,\pi) \stackrel{3}{\models} X\varphi] = [(M,s_1) \stackrel{3}{\models} \varphi]$$

$$[(M,\pi) \stackrel{3}{\models} \varphi_1 U \varphi_2] = \begin{cases} tt & \text{if } \exists k \geq 0 : [([(M,s_k) \stackrel{3}{\models} \varphi_2] = tt) \wedge (\forall j < k : [(M,s_j) \stackrel{3}{\models} \varphi_1] = tt)] \\ ff & \text{if } \forall k \geq 0 : [([(M,s_k) \stackrel{3}{\models} \varphi_2] = ff) \vee (\exists j < k : [(M,s_j) \stackrel{3}{\models} \varphi_1] = ff)] \\ \perp & \text{otherwise} \end{cases}$$

$$[(M,\pi) \stackrel{3}{\models} \varphi_1 V \varphi_2] = \begin{cases} tt & \text{if } \forall k \geq 0 : [(\forall j < k : [(M,s_j) \stackrel{3}{\models} \varphi_1] \neq tt) \Rightarrow ([(M,s_k) \stackrel{3}{\models} \varphi_2] = tt)] \\ ff & \text{if } \exists k \geq 0 : [(\forall j < k : [(M,s_j) \stackrel{3}{\models} \varphi_1] = ff) \wedge ([(M,s_k) \stackrel{3}{\models} \varphi_2] = ff)] \\ \perp & \text{otherwise} \end{cases}$$

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

Theorem 2.10 *[14] Let* $H \subseteq S_C \times S_A$ *be the 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) $[(M_A, s_a) \stackrel{3}{\models} \varphi] = tt \Rightarrow [(M_C, s_c) \models \varphi] = tt$*, and (2)* $[(M_A, s_a) \stackrel{3}{\models} \varphi] = ff \Rightarrow [(M_C, s_c) \models \varphi] = ff$*.*
We conclude that (1) $[M_A \stackrel{3}{\models} \varphi] = tt \Rightarrow [M_C \models \varphi] = tt$*, and (2)* $[M_A \stackrel{3}{\models} \varphi] = ff \Rightarrow [M_C \models \varphi] = ff$*.*

3 Using Games to Produce Annotated Counter-Examples

In this section we describe how to construct an *annotated counter-example* from the coloring of a game-graph for M and φ in case M does not satisfy φ.

First, the coloring algorithm 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 $\chi(n_0) = F$, the following DFS/BFS-like algorithm finds an *annotated counter-example*, denoted C, which is a subgraph of $G_{M\times\varphi}$ colored by F.

The algorithm `ComputeCounter`
Initially: $new = \{(s_0, \varphi)\}$, $C = \emptyset$.
while $new \neq \emptyset$
n = remove (new)

- if n was already handled - continue.
- if n is a terminal node - continue. $\backslash * \; sons = \emptyset \; * \backslash$
- if n is an $\vee$-node, then for each son n' of n add n' to new and (n, n') to C.
- if n is an $\wedge$-node, then add $cause(n)$ to new and $(n, cause(n))$ to C.

Complexity: Clearly, the construction of the annotated counter-example 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 $O(|M| \cdot |\varphi|)$.

The computed annotated counter-example can be viewed as the part of the winning strategy of the refuter that is sufficient to guarantee its victory. We formalize and prove this notion in the next section. Intuitively speaking, it is indeed a counter-example in the sense that it points out the reasons for φ's refutation on the model. Each node in C is marked by a state s and by a subformula φ_1, such that $\chi((s, \varphi_1)) = F$, thus by Theorem 2.6, $[s \models \varphi_1] =$ 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. Hence, by analyzing the annotated counter-example, one can understand why each subformula, and in particular the main formula, is refuted in the relevant state.

Note that, for the correctness of the annotated counter-example, 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. An example that demonstrates its importance appears in Appendix B.

3.1 Properties of the Annotated Counter-example

The annotated counter-example is a subgraph of the game-graph, and as such it has the properties of the game-graph. In addition, for each node $n \in C$, $\chi(n) = F$. Another important property is:

Lemma 3.1 *The annotated counter-example contains non-trivial SCCs* if and only if *at least one of the nodes in the SCC was colored due to a witness.*

Corollary 3.2 *Non-trivial SCCs in the annotated counter-example are either AU-SCCs or EU-SCCs.*

Corollary 3.2 results from Lemma 3.1, since only nodes in AU or EU SCCs are colored by F due to witness.

The property of the annotated counter-example described in Lemma 3.1, along with Corollary 3.2, imply that any non-trivial SCC that appears in the annotated counter-example indicates a refutation of the U operator, which results, at least partly, from an infinite path, where weak until is satisfied, but not strong until. This intuition results from the properties of the coloring algorithm. If a node is colored due to a witness, then finite information alone is not sufficient to cause its color. In the case of $A(\varphi_1 U \varphi_2)$, this means that there is no finite path where φ_1 stops 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 paths.

3.2 The Annotated Counter-Example is Sufficient and Minimal

In this section we first informally describe our requirements of a counter-example. We then formalize these requirements for annotated counter-examples and show that the result of algorithm `ComputeCounter` fulfills them. Generally speaking, for a sub-model to be a counter-example, 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.

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

Definition 3.3 *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.*

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 3.4 *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 it. In fact, one may notice that the colors of terminal nodes determine the coloring function of the full game-graph. Thus, to capture this notion, 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 $N \setminus A$.

We can now formalize the notion of an annotated counter-example.

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 counter-example *if it satisfies the following conditions. (1) For each node $n \in \tilde{C}$, $\chi(n) = F$; (2) $\tilde{C}$ is independent of G; and (3) $\tilde{C}$ is minimal.*

The first two requirements in Definition 3.5 imply that $\tilde{C}$ is *sufficient* for explaining why the initial node is colored by F. Therefore, it also explains why the formula is refuted by the model. 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, will be colored by F. The third condition shows that $\tilde{C}$ is also "necessary".

We now show that the result of algorithm `ComputeCounter`, denoted C, is indeed an annotated counter-example. The first requirement is obviously fulfilled, as described in Section 3.1. The following theorems prove that C satisfies the other two conditions as well.

Theorem 3.6 *C is independent of G.*

The correctness of Theorem 3.6 strongly depends on the choice of $cause(n)$ as the son of an $\wedge$-node in the algorithm `ComputeCounter`.

Theorem 3.7 *C is* minimal *in the sense that removing a node or an edge will result in a subgraph that is not independent of G.*

3.3 Practical Considerations

Since the annotated counter-example may be big and difficult to understand, several simplifications may be suggested. Each non-trivial MSCC can be replaced by a single node, annotated with its witness. Auxiliary edges may be collapsed, resulting in a sub-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. Note that we present a single annotated counter-example but can (interactively) give them all, using a variation of the algorithm `ComputeCounter`.

4 Game-Based Model Checking On Abstract Models

We suggest a generalization of the game-based model checking algorithm for evaluating a CTL formula φ over a KMTS M according to the 3-valued semantics. A discussion on solving the 3-valued problem by reducing it to two instances of the 2-valued problem, as suggested in [16], appears in Appendix C, where the advantages of the direct solution, described in this section, are presented.

We start with the description of the 3-valued game. 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 play 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 $AX\varphi_1$ should be checked upon may-transitions, but its falseness should be checked upon must-transitions.

The new moves of the game are:

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

Intuitively, the players use must-transitions in order to win, while they use may transitions in order to prevent the other player from winning. 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 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 are "consistent" in their moves:

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

The new winning criteria:

- $\forall$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 configuration $C_i = (s, \text{ff})$, 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.
- $\exists$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 configuration $C_i = (s, \text{tt})$, 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.

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. $[(M, s) \overset{3}{\models} \varphi] = tt$ *iff* $\exists$*loise has a winning strategy for the game starting at* (s, φ).
2. $[(M, s) \overset{3}{\models} \varphi] = ff$ *iff* $\forall$*belard has a winning strategy for the game starting at* (s, φ).
3. $[(M, s) \overset{3}{\models} \varphi] = \perp$ *iff none of the players has a winning strategy for the game starting at* (s, φ).

In order to use the above correspondence for model checking, we generalize the game-based model checking algorithm. The construction of the (3-valued) game-graph, denoted $G_{M\times\varphi}$, is defined as for the "concrete" game. 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 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').
- 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.

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 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 a sub-SCC of a may-SCC. As a result, we can have the graph partitioned to MSCCs based on may-edges, and each such may-MSCC can be further partitioned to MSCCs based on the must-edges in it. Lemma 2.4 holds for both types of SCCs in the 3-valued game-graph $G_{M\times\varphi}$. Thus, the notion of a *witness* in an SCC is also valid.

In practice, the type of a non-trivial MSCC of interest depends on the witness that is associated with it. For example, for an AU witness loops can only be used for refutation. To identify "real" loops, we need to use must-edges. Thus for such a witness, we need a must-MSCC. 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, we need a may-MSCC. Similarly, for an EU witness, we need a may-MSCC, whereas for an EV witness, a must-MSCC is used.

The (3-valued) coloring algorithm :

Partition: The game-graph is partitioned into its may-MSCCs. Each non-trivial may-MSCC with an AU or EV witness is further partitioned into its must-MSCCs, such that the original may-MSCC is *replaced* by its must-MSCCs. The resulting sets are denoted Q_i's. A partial order is determined on the Q_i's as follows.

Order: The initial partition to may-MSCCs induces an initial partial order such that transitions go out of a set only to itself or to a "smaller" set. All the must-MSCCs that result from the partitioning of a may-MSCC have a partial order between them such that *must* transitions go out of a set only to itself or to a "smaller" set. The combination of the initial partial order and the partial order within each may-MSCC result in a partial order between all the Q_i's. The partial order can be extended to a total order $\leq$ arbitrarily.

Coloring: The coloring algorithm consists of two phases, which are executed alternately.

1. **Sons-coloring phase:**
   Apply the following rules to *all* the nodes in $G_{M\times\varphi}$ 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 T; ***(F)*** if it has a must-son that is colored F; ***(?)*** if all its must sons are colored T or ? and it has a may-son that is colored F or ?.
   - An EX-node is colored by ***(T)*** if it has a must-son that is colored T; ***(F)*** if all its may-sons are colored F; ***(?)*** if it has a may-son that is colored T or ? and all its must-sons are colored F or ?.
   - An $\wedge$-node, other than AX-node, is colored by ***(T)*** if both its sons are colored T; ***(F)*** if it has a son that is colored F; ***(?)*** if it has a son that is colored ? and the other one is colored ? or T.
   - An $\vee$-node, other than EX-node, is colored by ***(T)*** if it has a son that is colored T; ***(F)*** if both its sons are colored F; ***(?)*** if it has a son that is colored ? and the other one is colored ? or F.
2. **Witness-coloring phase:**
   If after the propagation of the rules of phase 1 there are still nodes in $G_{M\times\varphi}$ that remain uncolored, then let Q_i be the smallest set with respect to $\leq$ that is not yet fully colored. Q_i must be either a non-trivial may-MSCC or a subgraph of such an MSCC that has exactly one witness (by Lemma 2.4). The uncolored nodes *in* Q_i are colored according to the witness in two phases, as follows.
   - The witness is of the form $A(\varphi_1 U \varphi_2)$ or $E(\varphi_1 U \varphi_2)$:
     (a) Repeatedly color ? each node in Q_i satisfying one of the following, until there is no change.
     - An $\wedge$-node (AX-node) that all its (must) sons are colored by T or ?.
     - An $\vee$-node (EX-node) that has a (may) son that is colored by T or ?.

     (b) Color the remaining nodes in Q_i by F.
   - The witness is of the form $A(\varphi_1 V \varphi_2)$ or $E(\varphi_1 V \varphi_2)$:
     (a) Repeatedly color ? each node in Q_i satisfying one of the following, until there is no change.
     - An $\wedge$-node (AX-node) that has a (may) son that is colored by F or ?.
     - An $\vee$-node (EX-node) that all its (must) sons are colored by F or ?.

     (b) Color the remaining nodes in Q_i by T.

   Note that only nodes from a single Q_i are colored in this phase.

The result of the coloring algorithm is a *3-valued coloring function* $\chi : N \to \{T, F, ?\}$. Note, that a node is colored ? only if there is evidence that it can no longer be colored otherwise. In other cases, another method is used to determine its color. Appendix D presents an example of a 3-valued game-graph and its coloring.

Theorem 4.3 *Let $G_{M\times\varphi}$ be a 3-valued game-graph and let n be a node in the game-graph, then:*

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

Implementation issues and Complexity: The coloring algorithm can be implemented in linear running time with respect to the size of the game-graph $G_{M\times\varphi}$, using a variation of an AND/OR graph, similarly to the algorithm described in [20] for checking the nonemptiness of the language of a simple weak alternating word automaton. Thus, its running time is bounded by $O(|M| \cdot |\varphi|)$.
As a conclusion of Theorem 4.2 and Theorem 4.3, we get the following theorem.

Theorem 4.4 *Let M be a KMTS and φ a CTL formula. Then, for each $n = (s, \varphi_1) \in G_{M\times\varphi}$:*

1. $[(M,s) \overset{3}{\models} \varphi_1] = tt$ *iff* $n = (s, \varphi_1)$ *is colored by* T.
2. $[(M,s) \overset{3}{\models} \varphi_1] = ff$ *iff* $n = (s, \varphi_1)$ *is colored by* F.
3. $[(M,s) \overset{3}{\models} \varphi_1] = \perp$ *iff* $n = (s, \varphi_1)$ *is colored by* $?$.

After coloring the 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 4.4 along with Theorem 2.10, 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 to the abstract model. Furthermore, if the result is ff, a *concrete* annotated counter-example can be produced, using an extension of the `ComputeCounter` algorithm, as described in Appendix E.

5 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. When the result is $\perp$, 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 counter-example as in [21, 7, 2, 9, 5], 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. Let $\gamma : S_A \rightarrow 2^{S_C}$ be the concretization function. Given the abstract 3-valued game-graph G, based on 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. The refinement is done by splitting abstract states according to criteria obtained from *failure nodes*. A node is a *failure node* if it is colored by $?$, whereas none of its sons is colored by $?$ at the time it gets colored by the algorithm. Such a node is a failure node in the sense that it can be seen as the point where the loss of information occurred. Note that a failure node may have uncolored sons at the time it was 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. The coloring algorithm is adapted to 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 $cont(n)$.

Searching For a Failure Node: A failure node is found by a DFS-like greedy algorithm, starting from n_0.

- If the current node n is a *failure node*, the algorithm ends.
- As long as n is not a failure node, the algorithm proceeds to $cont(n)$.

Lemma 5.1 *A failure node is a node colored by $?$, which is either a terminal node, or one of the following.*

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

Failure Analysis: Given a failure node n, it provides us with criteria for refinement. The refinement problem is reduced to the problem of separating sets of (concrete) states. This problem can be solved by known techniques, depending on the type of abstraction used (e.g. [9, 7]). 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 both concrete states that are labeled by l and by $\neg l$. The indefinite color is avoided by separating $\gamma(s_a)$ to two sets $\{s_c \in \gamma(s_a) : l \in L_C(s_c)\}$ and $\{s_c \in \gamma(s_a) : \neg l \in L_C(s_c)\}$.
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 $sons_K = \bigcup\{\gamma(s'_a) : (s'_a, \varphi_1) \in may\text{-}sons(n) \ \wedge \ \chi((s'_a, \varphi_1)) = K\}$ 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, and has a may-son $n' = (s'_a, \varphi_1)$ colored by ?. Let $conc_? = \gamma(s_a) \cap \{s_c \in S_C : \exists s'_c \in \gamma(s'_a), s_c \to s'_c\}$ 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. Consider case 2, where the failure node n is an AX-node. If $conc_F = \gamma(s_a)$, then every state represented by s_a has a refuting son. Thus, n can be colored by F instead of ?. If $conc_F = \emptyset$, then none of the concrete states in $\gamma(s_a)$ has a transition to a concrete state represented by the F-colored may-sons of n. Thus, the may-edges from n to such sons may be removed: none of them represents concrete transitions. Similar arguments apply to the rest of the cases as well. Either way, the game-graph can be recolored starting from the may-MSCC containing n.

The purpose of the split derived from cases 1-2 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. These new definite results can be used by the incremental algorithm, suggested below. We now consider case 3. Intuitively, 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 algorithm, 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 to obtain a must edge between (parts of) n and n'.

Theorem 5.2 *For finite concrete models, iterating the abstraction-refinement process is guaranteed to terminate with a definite answer.*

5.1 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 has a global effect. In practice, there is no reason to split states for which the model checking results are definite. The game-based model checking algorithm provides a convenient framework to use previous results. This leads to an *incremental* model checking algorithm based on iterative abstraction-refinement, where each iteration consists of abstraction, model checking and refinement. After each iteration, we now remember the (abstract) nodes colored by definite colors, as well as nodes for which a definite color was discovered during failure analysis. During the construction of a new refined game-graph, we prune the game-graph in nodes that are *sub-nodes* of nodes from previous iterations. A node (s_a, φ) is a *sub-node* of (s'_a, φ') if $\varphi = \varphi'$ and the set of concrete states represented by s_a is a subset of those represented by s'_a. As a result, only the reachable subgraph that was previously colored by ? is refined. The coloring algorithm considers the nodes where the game-graph was pruned as terminal nodes and colors them by their previous colors. Since previous runs use coarser abstractions, the number of nodes from previous runs should be much smaller than the number of refined nodes. Therefore, this pruning is worth-while.

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 [17, 32, 28, 15], this means that the abstract states "agree" on all the predicates that exist before the refinement.

6 Conclusion

In this work, we have exploited the game-theoretic approach of CTL model checking to produce annotated counter-examples 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 counter-example introduced in this paper may be seen as a minimal part of it that is sufficient to explain the result.

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 [18], 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 [20], as described in [24]. 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.

References

[1] A. Asteroth, C. Baier, and U. Assmann. Model checking with formula-dependent abstract models. In *Computer-Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 155–168, 2001.

[2] Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, July 2002.

[3] Benedikt Bollig, Martin Leucker, and Michael Weber. 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., 2002.

[4] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. *Lecture Notes in Computer Science*, 1877:168–182, 2000.

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

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

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

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

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

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

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

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

[13] D.Peled, A.Pnueli, and L.Zuck. From falsification to verification. In *FSTTCS*, volume 2245 of *LNCS*. Springer-Verlag, 2001.

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

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

[16] Patrice Godefroid and Radha Jagadeesan. On the expressiveness of 3-valued models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*, volume 2575 of *Lecture Notes in Computer Science*, pages 206–222. Springer-Verlag, January 2003.

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

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 58–70. ACM Press, 2002.

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

[20] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM)*, 47(2):312–360, 2000.

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

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

[23] Woohyuk Lee, Abelardo Pardo, Jae-Young Jang, Gary D. Hachtel, and Fabio Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD*, pages 76–81, 1996.

[24] Martin Leucker. 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)*, September 1999.

[25] Jorn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In *Computer Aided Verification*, pages 316–327, 1999.

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

[27] Kedar S. Namjoshi. Certifying model checkers. In *13th Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*. Springer-Verlag, 2001.

[28] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proc. of Conference on Computer-Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 435–449, Chicago, IL, USA, July 2000. Springer.

[29] Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional mu-calculus model checking. In *Computer Aided Verification*, pages 12–23, 1997.

[30] Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference*, pages 457–462, 1998.

[31] Doron Peled and Lenore Zuck. From model checking to a temporal proof. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 1–14. Springer-Verlag New York, Inc., 2001.

[32] Hassen Saidi and Natarajan Shankar. Abstract and model check while you prove. In *Proceedings of the eleventh International Conference on Computer-Aided Verification (CAV99)*, Jul 1999.

[33] Colin Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.

[34] Li Tan and Rance Cleaveland. Evidence-based model checking. In *14th Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 455–470. Springer Verlag, July 2002.

A Semantics of CTL formulae

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:

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

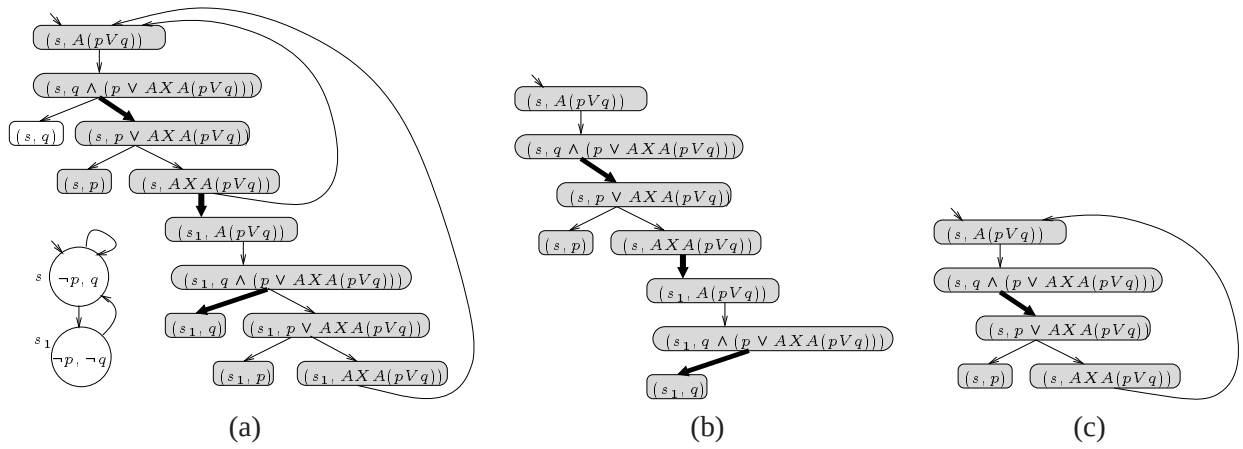

Figure 1: (a) A colored game-graph, where white nodes are colored by T, grey nodes are colored by F and bold edges point to the *cause* of an $\wedge$-node, (b) Its annotated counter-example and (c) A possible result of `ComputeCounter` without the use of the cause in $\wedge$-nodes.

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

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

B The Importance Of the Cause In Algorithm `ComputeCounter`

Figure 1 demonstrates the necessity of choosing $cause(n)$ as a son of an $\wedge$-node n when computing an annotated counter-example. Figure 1(a) presents a colored game-graph G, 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 1(b) presents the annotated counter-example computed by `ComputeCounter`, where it can be seen that the reason for refutation is the existence of the path $s, s_1, ..$ where q is not satisfied in s_1, although it was not released by p (p does not hold in s). On the other hand, Figure 1(c) presents a subgraph of the G, that is computed by a variation of `ComputeCounter`, where for an $\wedge$-node, an arbitrary son that is colored by F is chosen. In the example, the node $(s, A(pVq))$ was chosen as a 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 $s, s, ...$ However, this path satisfies pVq, such that it does not prove refutation. It can be shown that this subgraph is not independent of G.

C 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 3-valued semantics preserves both truth and falseness of a formula from the abstract model to the concrete one.

The definition of $[(M, s) \models \varphi]$ can be extended to a KMTS using a 2-valued semantics as well [12]. 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 by $[(M, s) \stackrel{2}{\models} \varphi]$. 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 C.1 *[12] Let $H \subseteq S_C \times S_A$ be a mixed simulation relation from M_C to 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 $[(M_C, s_c) \models \varphi] = tt$.*

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 game's purpose is also to prove φ's satisfaction. As such, $\exists$loise's moves in configurations with $EX\varphi'$ formulae need to use $\xrightarrow{\text{must}}$ transitions, since by the semantics definition, existential formulae are interpreted over *must*-paths. Similarly, $\forall$belard's moves in configurations with $AX\varphi'$ formulae need to use $\xrightarrow{\text{may}}$ 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 $\xrightarrow{\text{must}}$ is not necessarily total. Thus, a configuration of the form $(s, EX\varphi')$ may also be a terminal configuration, if s has no outgoing $\xrightarrow{\text{must}}$ transitions. A play that ends in such a configuration is won by $\forall$belard.

Clearly, the relation between the existence of winning strategies and satisfaction of the formula, as described in Theorem 2.3 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 2.5 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 next theorem.

Theorem C.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.

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. $O(|M| \cdot |\varphi|)$.

C.1 Application to 3-valued Model Checking

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

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

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

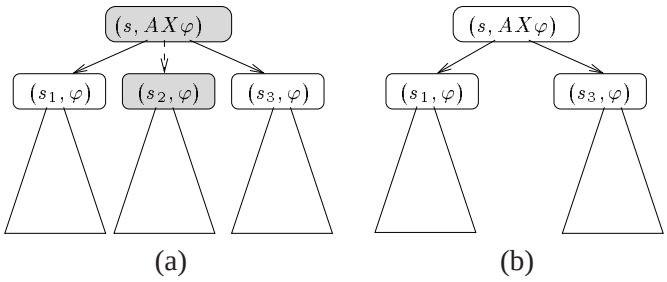

Figure 2: A satisfaction graph (a) versus a refutation graph (b) for $AX\varphi$

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

Thus, 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 in order to evaluate the 3-valued truth value of φ over M_A.

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 $\neg\varphi$ over M_A using the 2-valued semantics (with negations pushed to the literals). 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 counter-example 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 counter-example by the algorithm `ComputeCounter`. The abstract annotated counter-example is guaranteed not to be spurious and can be matched with a concrete one by a greedy algorithm, as described in Appendix E.

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 2, where in the satisfaction graph (a) the initial node $(s, AX\varphi)$ is colored by F 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), does not appear in the refutation graph, since it 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 "$[M \overset{3}{\models} \varphi]$?", as accurately as the direct 3-valued approach. However, for the refinement analysis we are interested in the *inner* nodes as well, that

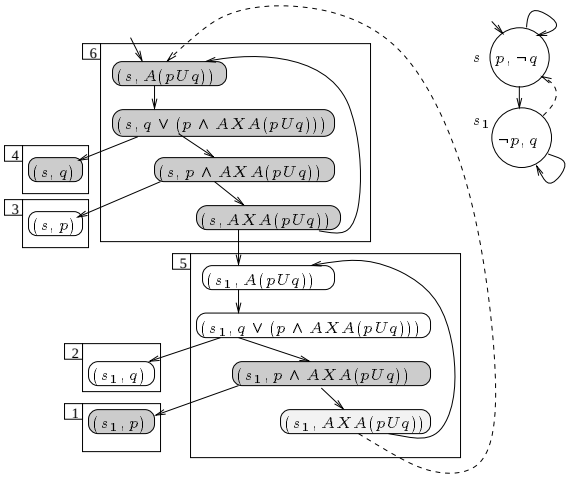

Figure 3: A colored 3-valued game-graph, where dashed edges are may-edges, solid edges are must-edges or auxiliary edges, and rectangles depict the partition of the nodes. White nodes are colored by T, dark grey nodes are colored by F, and light grey nodes are colored by $?$.

are not necessarily mutual to both the 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 [16] and then model checking the PKS under the 3-valued semantics by running a standard 2-valued model checker twice, as described in [4].

D 3-valued Coloring Example

Figure 3 presents a 3-valued game-graph G, where dashed edges represent may-edges and solid edges represent must-edges, as well as auxiliary edges. G has a single non-trivial may-MSCC with an AU-witness. Thus, it is partitioned into two must-MSCCs. The resulting partition of G to Q_i's is depicted by rectangles in Figure 3, where their numbers 1-6 determine the order $\leq$. The coloring algorithm starts from phase 1 by coloring the terminal nodes (in Q_1-Q_4). The node $(s_1, p \wedge AXA(pUq))$ is then colored by F (due to its son (s_1, p)). The node $(s_1, q \vee (p \wedge AXA(pUq)))$ is colored by T (due its son (s_1, q)), which causes the node $(s_1, A(pUq))$ to be colored by T as well. At this point, none of the remaining nodes can be colored. Thus, the algorithm proceeds to phase 2, where Q_5 is the smallest set with uncolored nodes, and its witness is of the form AU. The node $(s_1, AXA(pUq))$ is colored by $?$ in phase 2a, since it is an AX node and its only must son is colored by T. The algorithm then returns to phase 1, however, none of the rules is applicable. Thus, Q_6 is tackled in phase 2. None of its nodes can be colored by $?$. Thus, they are all colored by 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 evidence for refutation. The final coloring function can be seen in Figure 3.

E Constructing a Concrete Counter Example

In this section we show how to produce a concrete annotated counter-example from the 3-valued abstract game-graph that was used for model checking when 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, 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_A, based on the abstract model M_A, and its coloring function $\chi : N \rightarrow \{T, F, ?\}$,

such that $\chi(n_{0a}) = F$ for some initial node n_{0a}, we can use a variation of the algorithm `ComputeCounter` to produce an *abstract* annotated counter-example C_A. The difference is that in an AX-node we choose a *must-son* for the annotated counter-example and in an EX-node, we add all its *may-sons* to the annotated counter-example.

In order to find a *concrete* annotated counter-example, we need to replace each abstract state s_a, that represents a set of concrete states, with a single concrete state s_c from $\gamma(s_a)$. Since we are dealing with the annotated counter-example, some of the edges between nodes are auxiliary edges, that do not really represent advancements along transitions of the structure. If this is the case then the same concrete state should eventually match both these nodes. For an AX-node, the annotated counter-example shows one son that refutes the property. Given such a node n_a, and its only son in the counter-example n'_a, we need to match both their states with concrete states that have a concrete transition between them. For an EX-node, the annotated counter-example shows refutation in all its sons. Hence, given such a node n_a, we need to match its abstract state s_a with a concrete state s_c and add *all* its concrete sons to the concrete annotated counter-example.

Hence, the **concretization algorithm** of C_A for producing a concrete annotated counter-example, C_C, is described as follows.

- Choose the initial concrete node to be $n_{0c} = (s_{0c}, \varphi)$, where s_{0a} is the initial abstract state that appears in n_{0a} and s_{0c} is an arbitrary node from $\gamma(s_{0a}) \cap S_{0C}$.
- Apply the recursive procedure `ComputeSons` on $(n_{0c},\ n_{0a})$.

Given a concrete node $n_c = (s_c, \varphi')$ and the abstract node $n_a = (s_a, \varphi') \in C_A$ that matches it, the procedure `ComputeSons`(n_c, n_a) creates the concrete sons of n_c as follows:

- If $\varphi' = EX\varphi_1$, then for each state s'_c such that $s_c \rightarrow s'_c$, the node (s'_c, φ_1) is added to the concrete annotated counter-example as a son of n_c. Each such node matches an abstract node (s'_a, φ_1), such that $s'_c \in \gamma(s'_a)$ which is a son of n_a in the abstract annotated counter-example.
- If $\varphi' = AX\varphi_1$, then n_a has one son $n'_a = (s'_a, \varphi_1)$ in C_A. An arbitrary state s'_c is chosen from $\{s'_c \in S_C : s_c \rightarrow s'_c\} \cap \gamma(s'_a)$ and the node (s'_c, φ_1) is added to the concrete annotated counter-example as a son of n_c. The resulting son matches n'_a.
- If $\varphi' = \varphi_1 \vee \varphi_2$, then the nodes (s_c, φ_1) and (s_c, φ_2) are added to the concrete annotated counter-example as sons of n_c. They match the abstract nodes (s_a, φ_1) and (s_a, φ_2) respectively, which are both sons of n_a in C_A.
- If $\varphi' = \varphi_1 \wedge \varphi_2$, then n_a has one son $n'_a = (s_a, \varphi_i)$ in C_A, where $i \in \{1, 2\}$. The node (s_c, φ_i) is added to the concrete annotated counter-example as a son of n_c. The resulting son matches n'_a.

In any case, we then recursively call the procedure on the *new* concrete nodes (each one and the abstract node that it matches).

Basically, this is a greedy algorithm. The only situation where there is "freedom" in the choice of concrete states is in case of sons of AX-nodes. In EX-nodes the algorithm makes sure to include all the concrete sons in the annotated counter-example. As for other nodes, whose sons result from auxiliary edges, the algorithm makes sure to attach both the parent and the son with the same state.

Complexity: The running time of the concretization algorithm is linear in the size of the concrete annotated counter-example, 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 E.1 *The concretization algorithm does not fail.*

The following theorem guarantees the correctness of the concretization algorithm.

Theorem E.2 *C_C is an annotated counter-example for the concrete game-graph G_C, based on M_C and φ.*