Don't Know in the μ-Calculus

Orna Grumberg[1], Martin Lange[2], Martin Leucker[3], and Sharon Shoham[1]

[1] Computer Science Department, The Technion, Haifa, Israel
[2] Institut für Informatik, University of Munich, Germany
[3] Institut für Informatik, Technical University of Munich, Germany

Abstract. This work presents game-based model checking for abstract models with respect to specifications in μ-calculus, interpreted over a 3-valued semantics. If the model checking result is indefinite (*don't know*), the abstract model is refined, based on an analysis of the cause for this result. For finite concrete models our abstraction-refinement is fully automatic and guaranteed to terminate with a definite result *true* or *false*.

1 Introduction

This work presents a game-based [19] model checking approach for abstract models with respect to specifications in the μ-calculus, interpreted over a 3-valued semantics. In case the model checking result is indefinite (*don't know*), the abstract model is refined, based on an analysis of the cause for this result. If the concrete model is finite then our abstraction-refinement is fully automatic and guaranteed to terminate with a definite result (*true* or *false*).

Abstraction is one of the most successful techniques for fighting the state explosion problem in model checking [3]. Abstractions hide some of the details of the verified system, thus result in a smaller model. Usually, they are designed to be *conservative* for *true*, meaning that if a formula is true of the abstract model then it is also true of the concrete (precise) model of the system. However, if it is false in the abstract model then nothing can be deduced of the concrete one.

The *μ-calculus* [12] is a powerful formalism for expressing properties of transition systems using fixpoint operators. Many verification procedures can be solved by translating them into μ–calculus model checking [1]. Such problems include (fair) CTL model checking, LTL model checking, bisimulation equivalence and language containment of ω-regular automata.

In the context of abstraction, often only the universal fragment of μ-calculus is considered [14]. Over-approximated abstract models are used for verification of such formulae while under-approximated abstract models are used for their refutation.

Abstractions designed for full μ-calculus [6] have the advantage of handling both verification and refutation on the same abstract model. A greater advantage is obtained if μ-calculus is interpreted w.r.t the 3-valued semantics [11, 10]. This semantics evaluates a formula to either *true*, *false* or *indefinite*. Abstract models can then be designed to be conservative for both *true* and *false*. Only if the value of a formula in the abstract model is indefinite, its value in the concrete model is unknown. Then, a refinement is needed in order to make the abstract model more precise. Previous works [13, 16, 17] suggested abstraction-refinement mechanisms for various branching time logics over *2-valued* semantics.

Many algorithms for μ-calculus model checking with respect to the 2-valued semantics have been suggested [8, 20, 22, 5, 15]. An elegant solution to this problem is the game-based approach [19], in which two players, the verifier (denoted $\exists$) and the refuter (denoted $\forall$), try to win a game. A formula φ is true in a model M iff the verifier has a winning strategy, meaning that she can win any play, no matter what the refuter does. The game is played on a *game graph*, consisting of configurations $s \vdash \psi$, where s is a state of the model M and ψ is a subformula of φ. The players make moves between configurations in which they try to verify or refute ψ in s. These games can also be studied as *parity games* [7] and we follow this approach as well.

In model checking games for the 2-valued semantics, exactly one of the players has a winning strategy, thus the model checking result is either true or false. For the 3-valued semantics, a third value should also be possible. Following [18], we change the definition of a game for μ-calculus so that a *tie* is also possible.

To determine the winner, if there is one, we adapt the recursive algorithm for solving parity games by Zielonka [23]. This algorithm recursively computes the set of configurations in which one of the players has a winning strategy. It then concludes that in all other configurations the other player has a winning strategy.

In our algorithm we need to compute recursively three sets, since there are also those configurations in which none of the players has a winning strategy. We prove that our algorithm always terminates and returns the correct result.

In case the model checking game results in a tie, we identify a cause for the tie and try to eliminate it by refining the abstract model. More specifically, we adapt the presented algorithm to keep track of why a vertex in the game is classified as a tie. We then exploit the information gathered by the algorithm for refinement. The refinement is applied only to parts of the model from which tie is possible. Vertices from which there is a winning strategy for one of the players are not changed. Thus, the refined abstract models do not grow unnecessarily. If the concrete model is finite then our abstraction-refinement is guaranteed to terminate with a definite result.

It is the refinement based on the algorithm which rules out the otherwise interesting approach taken for example in [11, 10] in which a 3-valued μ-calculus model checking problem is reduced to two 2-valued μ-calculus model checking problems.

Organization of the paper The 3-valued μ-calculus is introduced in the next section. Then we describe the abstractions we have in mind. In Section 4, a 3-valued model-checking game for μ-calculus is shown. We give a model-checking algorithm for 3-valued games with a finite board in Section 5, and, explain how to refine the abstract model, in case of an indefinite answer in Section 6. We conclude in Section 7.

2 The 3-Valued μ-Calculus

Let $\mathcal{P}$ be a set of *propositional constants*, and $\mathcal{A}$ be a set of *action names*. Every $a \in \mathcal{A}$ is associated with a so-called *must*-action $a!$ and a *may*-action $a?$. Let $\mathcal{A}! = \{a! \mid a \in \mathcal{A}\}$ and $\mathcal{A}? = \{a? \mid a \in \mathcal{A}\}$. A *Kripke Modal Transition System* (KMTS) is a tuple $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{x} \mid x \in \mathcal{A}! \cup \mathcal{A}?\}, L)$ where $\mathcal{S}$ is a set of states, and $\xrightarrow{x} \subseteq \mathcal{S} \times \mathcal{S}$ for each $x \in \mathcal{A}! \cup \mathcal{A}?$ is a binary relation on states, s.t. for all $a \in Act$: $\xrightarrow{a!} \subseteq \xrightarrow{a?}$.

Let $\mathbb{B}_3 = \{\bot, ?, \top\}$ be partially ordered by $\bot \leq ? \leq \top$. Then $L : \mathcal{S} \to \mathbb{B}_3^{\mathcal{P}}$, where $\mathbb{B}_3^{\mathcal{P}}$ is the set of functions from $\mathcal{P}$ to $\mathbb{B}_3$. We use $\top$ to denote that a proposition holds in a state, $\bot$ for not holding, and ? if it cannot be determined whether it holds or not.

A Kripke structure in the usual sense can be regarded as a KMTS by setting $\xrightarrow{a!} = \xrightarrow{a?}$ for all $a \in \mathcal{A}$ and not distinguishing them anymore. Furthermore, its states labelling is over $\{\bot, \top\}$.

Let $\mathcal{V}$ be a set of propositional variables. Formulae of the *3-valued modal μ-calculus* in *positive normal form* are given by

$$\varphi \;::=\; q \;\mid\; \neg q \;\mid\; Z \;\mid\; \varphi \vee \varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \langle a \rangle \varphi \;\mid\; [a]\varphi \;\mid\; \mu Z.\varphi \;\mid\; \nu Z.\varphi$$

where $q \in \mathcal{P}$, $a \in \mathcal{A}$, and $Z \in \mathcal{V}$. Let 3-$\mathcal{L}_\mu$ denote the set of *closed* formulae generated by the above grammar, where the fixpoint quantifiers μ and ν are variable binders. We will also write η for either μ or ν. Furthermore we assume that formulae are well-named, i.e. no variable is bound more than once in any formula. Thus, every variable Z *identifies* a unique subformula $fp(Z) = \eta Z.\psi$ of φ, where the set $Sub(\varphi)$ of *subformulae* of φ is defined in the usual way.

Given variables Y, Z we write $Y \prec_\varphi Z$ if Z occurs freely in $fp(Y)$ in φ, and $Y <_\varphi Z$ if (Y, Z) is in the transitive closure of $\prec_\varphi$. The alternation depth $ad(\varphi)$ of φ is the length of a maximal $<_\varphi$-chain of variables in φ s.t. adjacent variables in this chain have different fixpoint types.

The semantics of a 3-$\mathcal{L}_\mu$ formula is an element of $\mathbb{B}_3^{\mathcal{S}}$—the functions from $\mathcal{S}$ to $\mathbb{B}_3$—which forms a boolean lattice when equipped with the following partial order: let $f, g : \mathcal{S} \to \mathbb{B}_3$. $f \sqsubseteq g$ iff $\forall s \in \mathcal{S} : f(s) \leq g(s)$. Joins (meets) in this lattice are denoted by $f \sqcup g$ ($f \sqcap g$, resp.). The complement of f, written $\overline{f}$ is defined by $\overline{f}(s) := \overline{f(s)}$ for $s \in \mathcal{S}$ where $\bot$ and $\top$ are complementary to each other, and $\overline{?} = ?$.

Then the *semantics* $[\![\varphi]\!]_\rho^{\mathcal{T}}$ of a 3-$\mathcal{L}_\mu$ formula φ w.r.t. a KMTS $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{x} \mid x \in \mathcal{A}! \cup \mathcal{A}?\}, L)$ and an *environment* $\rho : \mathcal{V} \to \mathbb{B}_3^{\mathcal{S}}$, which explains the meaning of free variables in φ, is an element of $\mathbb{B}_3^{\mathcal{S}}$. We assume $\mathcal{T}$ to be fixed and do not mention it explicitly anymore. With $\rho[Z \mapsto f]$ we denote the environment that maps Z to f and agrees with ρ on all other arguments. Later, when only closed formulae are considered, we will also drop the environment from the semantic brackets.

$$\begin{aligned}
[\![q]\!]_\rho &:= \lambda s.L(s)(q) \\
[\![\neg q]\!]_\rho &:= \lambda s.\overline{L(s)(q)} \\
[\![Z]\!]_\rho &:= \rho(Z)
\end{aligned}$$

$$
\begin{array}{lcl}
\llbracket \varphi \vee \psi \rrbracket_\rho & := & \llbracket \varphi \rrbracket_\rho \sqcup \llbracket \psi \rrbracket_\rho \\
\llbracket \varphi \wedge \psi \rrbracket_\rho & := & \llbracket \varphi \rrbracket_\rho \sqcap \llbracket \psi \rrbracket_\rho \\
\llbracket \langle a \rangle \varphi \rrbracket_\rho & := & \lambda s. \begin{cases} \top \text{ , if } \exists t \in \mathcal{S}, \text{ s.t. } s \xrightarrow{a!} t \text{ and } \llbracket \varphi \rrbracket_\rho (t) = \top \\ \bot \text{ , if } \forall t \in \mathcal{S}, \text{ if } s \xrightarrow{a?} t \text{ then } \llbracket \varphi \rrbracket_\rho (t) = \bot \\ ? \text{ , otherwise} \end{cases} \\
\llbracket [a] \varphi \rrbracket_\rho & := & \lambda s. \begin{cases} \top \text{ , if } \forall t \in \mathcal{S}, \text{ if } s \xrightarrow{a?} t \text{ then } \llbracket \varphi \rrbracket_\rho (t) = \top \\ \bot \text{ , if } \exists t \in \mathcal{S}, \text{ s.t. } s \xrightarrow{a!} t \text{ and } \llbracket \varphi \rrbracket_\rho (t) = \bot \\ ? \text{ , otherwise} \end{cases} \\
\llbracket \mu Z.\varphi \rrbracket_\rho & := & \bigsqcap \{ f \mid \llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} \sqsubseteq f \} \\
\llbracket \nu Z.\varphi \rrbracket_\rho & := & \bigsqcup \{ f \mid f \sqsubseteq \llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} \}
\end{array}
$$

Note that $s \xrightarrow{a!} t$ implies $s \xrightarrow{a?} t$.

The functionals $\lambda f.\llbracket \varphi \rrbracket_{\rho[Z \mapsto f]} : \mathbb{B}_3^{\mathcal{S}} \to \mathbb{B}_3^{\mathcal{S}}$ are monotone w.r.t. $\sqsubseteq$ for any Z, φ and $\mathcal{S}$. According to [21], least and greatest fixpoints of these functionals exist.

Approximants of 3-$\mathcal{L}_\mu$ formulae are defined in the usual way: if $fp(Z) = \mu Z.\varphi$ then $Z^0 := \lambda s.\bot$, $Z^{\alpha+1} := \llbracket \varphi \rrbracket_{\rho[Z \mapsto Z^\alpha]}$ for any ordinal α and any environment ρ, and $Z^\lambda := \bigsqcap_{\alpha<\lambda} Z^\alpha$ for a limit ordinal λ. Dually, if $fp(Z) = \nu Z.\varphi$ then $Z^0 := \lambda s.\top$, $Z^{\alpha+1} := \llbracket \varphi \rrbracket_{\rho[Z \mapsto Z^\alpha]}$, and $Z^\lambda := \bigsqcup_{\alpha<\lambda} Z^\alpha$.

Theorem 1. [21] *For all KMTS $\mathcal{T}$ with state set $\mathcal{S}$ there is an $\alpha \in \mathbb{O}\mathrm{rd}$ s.t. for all $s \in \mathcal{S}$ we have: if $\llbracket \eta Z.\varphi \rrbracket_\rho (s) = x$ then $Z^\alpha(s) = x$.*

3 Abstraction

We use *Kripke Modal Transition Systems* [11, 9] as abstract models that preserve satisfaction and falsification of 3-$\mathcal{L}_\mu$ formulae.

Let $\mathcal{T}_C = (\mathcal{S}_C, \{\xrightarrow{a}_C \mid a \in \mathcal{A}\}, L_C)$ be a (concrete) Kripke structure. Let $\mathcal{S}_A$ be a set of *abstract states* and $\gamma : \mathcal{S}_A \to 2^{\mathcal{S}_C}$ a total *concretization function* that maps each abstract state to the set of concrete states it represents. An abstract model, a KMTS $\mathcal{T}_A = (\mathcal{S}_A, \{\xrightarrow{x}_A \mid x \in \mathcal{A}! \cup \mathcal{A}?\}, L_A)$, can then be defined as follows.

The labelling of an abstract state is defined in accordance with the labelling of all the concrete states it represents. For $p \in \mathcal{P} : L_A(s_a)(p) = \top$ $(\bot)$ only if $\forall s_c \in \gamma(s_a) : L_C(s_c)(p) = \top$ $(\bot)$. In the remaining cases $L_A(s_a)(p) = ?$.

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

$$\frac{s \vdash \psi_0 \vee \psi_1}{s \vdash \psi_i} \ \exists: \ i \in \{0,1\} \qquad \frac{s \vdash \psi_0 \wedge \psi_1}{s \vdash \psi_i} \ \forall: \ i \in \{0,1\}$$

$$\frac{s \vdash \eta Z.\varphi}{s \vdash Z} \ \exists \qquad \frac{s \vdash Z}{s \vdash \varphi} \ \exists: \text{if } fp(Z) = \eta Z.\varphi$$

$$\frac{s \vdash \langle a \rangle \varphi}{t \vdash \varphi} \ \exists: \ s \xrightarrow{a!} t \text{ or } s \xrightarrow{a?} t \qquad \frac{s \vdash [a] \varphi}{s \vdash \varphi} \ \forall: \ s \xrightarrow{a!} t \text{ or } s \xrightarrow{a?} t$$

Fig. 1. The model checking game rules for 3-$\mathcal{L}_\mu$.

Theorem 2. [9] *Let $\mathcal{T}$ be a Kripke structure and let $\mathcal{T}'$ be a KMTS obtained from $\mathcal{T}$ with the abstraction process described above. Let s be a state of $\mathcal{T}$ and s' its corresponding abstract state in $\mathcal{T}'$. For all closed $\varphi \in$ 3-$\mathcal{L}_\mu$: $[\![\varphi]\!]^{\mathcal{T}'}(s') \neq ?$ implies $[\![\varphi]\!]^{\mathcal{T}}(s) = [\![\varphi]\!]^{\mathcal{T}'}(s')$.*

4 Model Checking Games for 3-$\mathcal{L}_\mu$

The *model checking game* $\Gamma_{\mathcal{T}}(s_0, \varphi_0)$ on a KMTS $\mathcal{T}$ with state set $\mathcal{S}$, initial state $s_0 \in \mathcal{S}$ and a 3-$\mathcal{L}_\mu$ formula φ_0 is played by players $\exists$ and $\forall$ in order to determine the truth value of φ_0 in s_0, cf. [19]. Configurations are elements of $\mathcal{C} \subseteq \mathcal{S} \times Sub(\varphi_0)$, and written $t \vdash \psi$. Each play of $\Gamma_{\mathcal{T}}(s_0, \varphi_0)$ is a maximal sequence of configurations that starts with $s_0 \vdash \varphi_0$. The game rules are presented in Figure 1. Each rule is marked by $\exists$ / $\forall$ to indicate which player makes the move. A rule is applied when the player is in configuration C_i, which is of the form of the upper part of the rule. C_{i+1} is then the configuration in the lower part of the rule. The rules shown in the first and third lines present a choice which the player can make. Since no choice is possible when applying the rules shown in the second line, we arbitrarily assign one player, let us say $\exists$, and call the rules *deterministic*. If no rule can be applied the play terminates.

Definition 1. *A play is called* $\exists$-consistent, *resp.* $\forall$-consistent, *if Player $\exists$, resp. Player $\forall$, never chooses a transition of type $\xrightarrow{a?}$ for some $a \in \mathcal{A}$.*

Player $\exists$ *wins* an $\exists$-consistent play $C_0, C_1, \ldots$ iff

1. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash q$ with $L(t)(q) = \top$ or $C_n = t \vdash \neg q$ with $L(t)(q) = \bot$, or
2. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash [a]\psi$ and there is no $t' \in \mathcal{S}$ s.t. $t \xrightarrow{a?} t'$, or
3. the outermost variable that occurs infinitely often is of type ν.

Player $\forall$ *wins* a $\forall$-consistent play $C_0, C_1 \ldots$ iff

4. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash q$ with $L(t)(q) = \bot$ or $C_n = t \vdash \neg q$ with $L(t)(q) = \top$, or
5. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash \langle a \rangle\psi$ and there is no $t' \in \mathcal{S}$ s.t. $t \xrightarrow{a?} t'$, or
6. the outermost variable that occurs infinitely often is of type μ.

In all other cases, the result of the play is a *tie*.

Definition 2. *The* truth value *of a configuration* $t \vdash \psi$ *in the context of* ρ *is the value of* $[\![\psi]\!]_\rho(t)$. *The value* $\top$ improves *both ? and* $\bot$, *while ? only improves* $\bot$. *On the other hand,* x worsens y *iff* y *improves* x.

An inspection of game rules and semantics shows: The deterministic rules preserve the truth value in a move from one configuration to another. Player $\exists$ cannot improve it but can preserve $\top$. Player $\forall$ cannot worsen it but can preserve $\bot$.

A *strategy* for player p is a partial function $\zeta : \mathcal{C} \rightarrow \mathcal{C}$, such that its domain is the set of configurations where player p moves. Player p plays a game according to a strategy ζ if all his choices agree with ζ. A strategy for player p is called a *winning strategy* if player p wins every play where he plays according to this strategy.

Theorem 3. *Given a KMTS* $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{x} \mid x \in \mathcal{A}! \cup \ Act?\}, L)$, *an* $s \in \mathcal{S}$, *and a closed* $\varphi \in 3\text{-}\mathcal{L}_\mu$, *we have:*

(a) $[\![\varphi]\!]^{\mathcal{T}}(s) = \top$ *iff Player* $\exists$ *has a winning strategy for* $\Gamma_{\mathcal{T}}(s, \varphi)$,
(b) $[\![\varphi]\!]^{\mathcal{T}}(s) = \bot$ *iff Player* $\forall$ *has a winning strategy for* $\Gamma_{\mathcal{T}}(s, \varphi)$,
(c) $[\![\varphi]\!]^{\mathcal{T}}(s) = ?$ *iff neither Player* $\exists$ *nor Player* $\forall$ *has a winning strategy for* $\Gamma_{\mathcal{T}}(s, \varphi)$.

Theorem 4. *Let* $\mathcal{T} = (\mathcal{S}, \{\xrightarrow{x} \mid x \in \mathcal{A}\}, L)$ *be a Kripke structure with* $s \in \mathcal{S}$ *and* $\mathcal{T}' = (\mathcal{S}', \{\xrightarrow{x} \mid x \in \mathcal{A}! \cup \mathcal{A}?\}, L')$ *be an abstraction of* $\mathcal{T}$ *with concretization function* γ. *Let* $s' \in \mathcal{S}'$ *with* $s \in \gamma(s')$.

(a) If Player $\exists$ *has a winning strategy for* $\Gamma_{\mathcal{T}'}(s', \varphi)$ *then* $\mathcal{T}, s \models \varphi$.
(b) If Player $\forall$ *has a winning strategy for* $\Gamma_{\mathcal{T}'}(s', \varphi)$ *then* $\mathcal{T}, s \not\models \varphi$.

5 Winning Model Checking Games for 3-$\mathcal{L}_\mu$

The previous section relates model checking games with the semantics of 3-$\mathcal{L}_\mu$. An algorithm estimating the winner of the game and a winning strategy is yet to be given. Note that the result of the previous section also holds for infinite-state systems. From now on, however, we restrict to finite KMTS.

For the sake of readability we will deal with parity games. Instead of Player $\exists$ and $\forall$, we talk of Player 0 and Player 1, resp., and use σ to denote Player 0 or 1 and $\bar{\sigma} = 1 - \sigma$ for the opponent.[1]

Parity games are traditionally used to describe the model checking game for μ-calculus. In order to describe our game for the 3-$\mathcal{L}_\mu$, we need to generalize them in the following way: (1) we have two types of edges: must edges and may edges, where every must edge is also a may edge, (2) terminal configurations (dead-end) are classified as either winning for one player, or as tie-configurations, and (3) a consistency requirement is added to the winning conditions.

[1] The numbers 0 and 1 have parities and this is more intuitive for this notion of game.

A *generalized parity game* $G = (A, \chi)$ has an *arena* $A = (V_0, V_1, V_{tie}, \xrightarrow{must}, \xrightarrow{may})$ for which every $v \in V_{tie}$ is a dead-end and $\xrightarrow{must} \subseteq \xrightarrow{may}$. The set of vertices is denoted by $V = V_0 \uplus V_1 \uplus V_{tie}$. $\chi : V \to \mathbb{N}$ is a *priority function* that maps each vertex $v \in V$ to a *priority*.

A play is a maximal sequence of vertices $v_0, \ldots$, where Player σ moves from v_i to v_{i+1} when $v_i \in V_\sigma$ and $(v_i, v_{i+1}) \in \xrightarrow{may}$. It is called *$\sigma$-consistent* iff Player σ chooses only moves that are (also) in $\xrightarrow{must}$. A σ-consistent play is *winning* for Player σ if

- it is finite and ends in V_σ, or
- it is infinite and the maximal priority occurring infinitely often is even when $\sigma = 0$ or odd when $\sigma = 1$.

All other plays are a *tie*.

A model checking game is a generalized parity game (see also [7]): Set V_0 to the configurations in which $\exists$ moves together with configurations in which the play terminates and $\exists$ wins. Set V_1 to the configurations in which $\forall$ moves, together with configurations in which the play terminates and $\forall$ wins. The remaining configurations, i.e. the ones of the form $t \vdash q$ or $t \vdash \neg q$ with $L(t)(q) = L(t)(\neg q) = ?$ are set to V_{tie}. $\xrightarrow{must}$ comprises the moves based on the rules shown in the first two lines in Figure 1 or when a $a!$-transition is taken while $\xrightarrow{may}$ comprises all possible moves. The priority of a vertex $t \vdash \varphi$ is only non-zero when φ is a fixpoint formula. Then, it is given by the alternation depth of φ, possibly plus 1 to assure that it is even iff the outermost fixpoint variable in φ is ν. It is easy to see that the notions of winning and winning strategies for both notions of games coincide.

We define an algorithm for solving generalized parity games. Our algorithm partitions V into three sets: W_0, W_1, W_{tie}, where for $\sigma \in \{0, 1\}$, the set W_σ consists of all the vertices from which Player σ has a winning strategy and the set W_{tie} consists of all the vertices from which none of the players has a winning strategy. When applied to model checking whether $s_0 \models \varphi_0$, we check when the algorithm terminates whether $v = s_0 \vdash \varphi_0$ is in W_0, W_1, or W_{tie} and conclude *true*, *false*, or *indefinite*, respectively.

We adapt the recursive algorithm for solving parity games by Zielonka [23]. Its recursive nature makes it easy to understand and analyze, allows simple correctness proofs, and can be used as basis for refinement.

The main idea of the algorithm presented in [23] is as follows. In each recursive call, σ denotes the parity of the maximal priority in the current game. The algorithm computes the set $W_{\bar{\sigma}}$ iteratively and the remaining vertices form W_σ. In our generalized game, we again compute $W_{\bar{\sigma}}$ iteratively, but we then add a phase where we also compute W_{tie} iteratively. Only then, we set W_σ to the remaining vertices.

We start with some definitions. For $X \subseteq V$, the subgraph of G induced by X, denoted by $G[X]$, is $(A|_X, \chi|_X)$ where $A|_X = (V_0', V_1', V_{tie} \cap X, \xrightarrow{must} \cap X \times X, \xrightarrow{may} \cap X \times X)$ and $\chi|_X$ is the restriction of χ to X. For $\sigma \in \{0, 1\}$, let B_σ denote the set of non-dead-end vertices that belong to V_σ in G, but become dead-ends in

$G[X]$. Then, in $G[X]$, $V'_\sigma = ((V_\sigma \setminus B_\sigma) \cup B_{\bar\sigma}) \cap X$. That is, vertices that become dead-ends, move to the set of vertices of the other player.

$G[X]$ is a *subgame* of G *w.r.t.* σ, for $\sigma \in \{0,1\}$, if all non-dead-end vertices of V_σ in G remain non-dead-ends in $G[X]$. It is a *subgame* of G if it is a subgame w.r.t. to both players. That is, if $G[X]$ is a subgame, then every dead-end in it is also a dead-end in G.

For $\sigma \in \{0,1\}$ and $X \subseteq V$, we define the must-attractor set $\text{Attr!}_\sigma(G,X) \subseteq V$ and the may-attractor set $\text{Attr?}_\sigma(G,X) \subseteq V$ of Player σ in G.

The *must-attractor* $\text{Attr!}_\sigma(G,X) \subseteq V$ is the set of vertices from which Player σ has a strategy in the game G to attract the play to X or a dead-end in V_σ while maintaining consistency. The *may-attractor* $\text{Attr?}_\sigma(G,X) \subseteq V$ is the set of vertices from which Player σ has a strategy in G to either (1) attract the play to X or a dead-end in $V_\sigma \cup V_{tie}$, possibly without maintaining his own consistency or (2) to prevent $\bar\sigma$ from playing consistently. In other words, if $\bar\sigma$ plays consistently, σ can attract the play to one of the vertices described in (1).

Let D_0, D_1, D_{tie} denote the dead-end vertices of V_0, V_1, V_{tie} respectively (i.e., $D_{tie} = V_{tie}$). It can be shown that the following is an equivalent definition of the sets $\text{Attr!}_\sigma(G,X)$ and $\text{Attr?}_\sigma(G,X)$.

$$\begin{aligned}
\text{Attr!}^0_\sigma(G,X) &= X \cup D_\sigma \\
\text{Attr!}^{i+1}_\sigma(G,X) &= \text{Attr!}^i_\sigma(G,X) \\
&\quad \cup \{v \in V_\sigma \setminus D_\sigma \mid \exists v'.v \xrightarrow{must} v' \wedge v' \in \text{Attr!}^i_\sigma(G,X)\} \\
&\quad \cup \{v \in V_{\bar\sigma} \setminus D_{\bar\sigma} \mid \forall v'.v \xrightarrow{may} v' \implies v' \in \text{Attr!}^i_\sigma(G,X)\} \\
\text{Attr!}_\sigma(G,X) &= \bigcup\{\text{Attr!}^i_\sigma(G,X) \mid i \geq 0\}
\end{aligned}$$

$$\begin{aligned}
\text{Attr?}^0_\sigma(G,X) &= X \cup D_\sigma \cup D_{tie} \\
\text{Attr?}^{i+1}_\sigma(G,X) &= \text{Attr?}^i_\sigma(G,X) \\
&\quad \cup \{v \in V_\sigma \setminus D_\sigma \mid \exists v'.v \xrightarrow{may} v' \wedge v' \in \text{Attr?}^i_\sigma(G,X)\} \\
&\quad \cup \{v \in V_{\bar\sigma} \setminus D_{\bar\sigma} \mid \forall v'.v \xrightarrow{must} v' \implies v' \in \text{Attr?}^i_\sigma(G,X)\} \\
\text{Attr?}_\sigma(G,X) &= \bigcup\{\text{Attr?}^i_\sigma(G,X) \mid i \geq 0\}
\end{aligned}$$

The latter definition of the attractor sets provides a method for computing them. As i increases, we calculate $\text{Attr!}^i_\sigma(G,X)$ or $\text{Attr?}^i_\sigma(G,X)$ until it is the same as $\text{Attr!}^{i-1}_\sigma(G,X)$ or $\text{Attr?}^{i-1}_\sigma(G,X)$, respectively.

Note that $\text{Attr!}^i_\sigma(G,X) \subseteq \text{Attr?}^i_\sigma(G,X)$, and that for $X' = V \setminus \text{Attr?}_\sigma(G,X)$ we have $X' = \text{Attr!}_{\bar\sigma}(G,X')$. Thus, the corresponding must and may attractors partition V.

Solving the Game

We present a recursive algorithm `SolveGame`(G) (see Algorithm 3) that computes the sets W_0, W_1, and W_{tie} for a parity game G. Let n be the maximum priority occurring in G.

$n = 0$:
$$\begin{aligned}
W_1 &= \text{Attr!}_1(G,\emptyset) \\
W_0 &= V \setminus \text{Attr?}_1(G,\emptyset) \\
W_{tie} &= \text{Attr?}_1(G,\emptyset) \setminus \text{Attr!}_1(G,\emptyset)
\end{aligned}$$

Algorithm 1 Winning vertices for the opponent: `ComputeOpponentWin`

1 **Function** ComputeOpponentWin(G, σ, n)
2 $W_{\bar{\sigma}} := \emptyset$.
3 **repeat**
4 $W'_{\bar{\sigma}} := W_{\bar{\sigma}}$
5 $X_{\bar{\sigma}} := \text{Attr!}_{\bar{\sigma}}(G, W_{\bar{\sigma}})$
6 $X_{\sigma} := V \setminus X_{\bar{\sigma}}$
7 $N := \{v \in X_{\sigma} \mid \chi(v) = n\}$
8 $Y := X_{\sigma} \setminus \text{Attr?}_{\sigma}(G[X_{\sigma}], N)$
9 $(Y_0, Y_1, Y_{tie}) := \text{SolveGame}(G[Y])$
10 $W_{\bar{\sigma}} := X_{\bar{\sigma}} \cup Y_{\bar{\sigma}}$
11 **until** $W'_{\bar{\sigma}} = W_{\bar{\sigma}}$
12 **return** $W_{\bar{\sigma}}$

Since the maximum priority of G is 0, Player 1 can only win G on dead-ends in V_1 or vertices from which he can consistently attract the play to such a dead-end. This is exactly $\text{Attr!}_1(G, \emptyset)$. From the rest of the vertices Player 1 does not have a winning strategy. For vertices in $V \setminus \text{Attr?}_1(G, \emptyset)$, Player 0 can always avoid reaching dead-ends in $V_1 \cup V_{tie}$, while playing consistently. Since the maximum priority in this subgraph is 0, it is easy to see that she wins in such vertices. The remaining vertices in $\text{Attr?}_1(G, \emptyset) \setminus \text{Attr!}_1(G, \emptyset)$ are a subset of $\text{Attr?}_1(G, \emptyset)$, which is why Player 0 does not win from them (and neither does Player 1, as previously claimed). Therefore none of the players wins in $\text{Attr?}_1(G, \emptyset) \setminus \text{Attr!}_1(G, \emptyset)$.

$\boldsymbol{n \geq 1}$**:** We assume that we can solve every game with maximum priority smaller than n. Let $\sigma = n \mod 2$ be the player that wins if the play visits infinitely often the maximum priority n.

We first compute $W_{\bar{\sigma}}$ in G. This is done by the function `ComputeOpponentWin` shown in Algorithm 1.

Intuitively, in each iteration we hold a subset of the winning region of Player $\bar{\sigma}$. We first extend it to $X_{\bar{\sigma}}$ by using the must-attractor set of Player $\bar{\sigma}$ (which ensures his consistency, line 5). From the remaining vertices, we disregard those from which Player σ can attract the play to a vertex with maximum priority n, perhaps by giving up his consistency. Left are the vertices in Y (line 8) and Player σ is basically trapped in it. He can only "escape" from it to $X_{\bar{\sigma}}$. Thus, we can add the winning region of Player $\bar{\sigma}$ in $G[Y]$ to his winning region in G. This way, each iteration results in a better (bigger) under approximation of the winning region of Player $\bar{\sigma}$ in G, until the full region is found (line 11). The correctness proof of the algorithm is sketched in the following.

Lemma 1. *1. For every X_{σ} as used in Algorithm 1, $G[X_{\sigma}]$ is a subgame w.r.t. σ.*

2. For every Y as used in Algorithm 1, $G[Y]$ is a subgame.

Moreover, the maximum priority in $G[Y]$ is smaller than n, which is why the recursion terminates.

Lemma 2. *At the beginning of each iteration in Algorithm 1, $W_{\bar{\sigma}}$ is a winning region for Player $\bar{\sigma}$ in G.*

Proof. The proof is by induction. The base case is when $W_{\bar{\sigma}} = \emptyset$ and the claim holds. Suppose that at the beginning of the ith iteration $W_{\bar{\sigma}}$ is a winning region for Player $\bar{\sigma}$ in G. We show that it continues to be so at the end of the iteration and therefore at the beginning of the $i+1$ iteration.

Clearly, $X_{\bar{\sigma}} = \mathrm{Attr!}_{\bar{\sigma}}(G, W_{\bar{\sigma}})$ is also a winning region for Player $\bar{\sigma}$ in G: by simply using his strategy to attract the play to $D_{\bar{\sigma}}$ or to $W_{\bar{\sigma}}$ (where he wins) while being consistent, and from there using the winning strategy of $W_{\bar{\sigma}}$ in G.

We now show that $Y_{\bar{\sigma}}$ is also a winning region of Player $\bar{\sigma}$ in G. We know that it is a winning region for him in $G[Y]$ (by the correctness of the algorithm `SolveGame` for games with a maximum priority smaller than n). As for G, for every vertex in $Y_{\bar{\sigma}}$, as long as the play remains in Y, Player $\bar{\sigma}$ can use his strategy for $G[Y]$. Since $G[Y]$ is a subgame, Player $\bar{\sigma}$ will always be able to stay within Y in his moves in G and if the play stays there, then he wins (since he uses his winning strategy). Clearly Player σ cannot move from Y to $X_\sigma \setminus Y = \mathrm{Attr?}_\sigma(G[X_\sigma], N)$. Otherwise the vertex $v \in Y \subseteq X_\sigma$ where this is done belongs to $\mathrm{Attr?}_\sigma(G[X_\sigma], \mathrm{Attr?}_\sigma(G[X_\sigma], N))$ (because the same move is possible in $G[X_\sigma]$). Hence v belongs to $\mathrm{Attr?}_\sigma(G[X_\sigma], N)$ as well, in contradiction to $v \in Y$. Finally, if Player σ moves to $V \setminus X_\sigma = X_{\bar{\sigma}}$, then Player $\bar{\sigma}$ will use his strategy for $X_{\bar{\sigma}}$ in G and also win.

We conclude that $X_{\bar{\sigma}} \cup Y_{\bar{\sigma}}$ is a winning region for Player $\bar{\sigma}$ in G. □

This lemma ensures that the final result $W_{\bar{\sigma}}$ of `ComputeOpponentWin` is indeed a subset of the winning region of Player $\bar{\sigma}$ in G. It remains to show that this is actually an equality, i.e. that no winning vertices are missing.

Lemma 3. *When $W'_{\bar{\sigma}} = W_{\bar{\sigma}}$, then $V \setminus W_{\bar{\sigma}}$ is a non-winning region for Player $\bar{\sigma}$ in G.*

Proof. When $W'_{\bar{\sigma}} = W_{\bar{\sigma}}$, it must be the case that the last iteration of `SolveGame` ended with $Y_{\bar{\sigma}} = \emptyset$, and $W_{\bar{\sigma}} = X_{\bar{\sigma}}$. Therefore it suffices to show that $V \setminus X_{\bar{\sigma}} = X_\sigma$ is a non-winning region for Player $\bar{\sigma}$ in G.

Clearly, Player $\bar{\sigma}$ cannot move from X_σ to $X_{\bar{\sigma}}$ without compromising his consistency. Otherwise the vertex $v \in X_\sigma$ where this is done belongs to $\mathrm{Attr!}_{\bar{\sigma}}(G, X_{\bar{\sigma}})$ and so to $X_{\bar{\sigma}}$ as well. This contradicts $v \in X_\sigma$. Hence, Player $\bar{\sigma}$ cannot win by moving to $X_{\bar{\sigma}}$. As $G[X_\sigma]$ is a subgame w.r.t. σ, Player σ is never obliged to move to $X_{\bar{\sigma}}$.

Consider the case where the play stays in X_σ. In order to prevent Player $\bar{\sigma}$ from winning, Player σ will play as follows. If the current configuration is in Y, then Player σ will use his strategy on $G[Y]$ for preventing Player $\bar{\sigma}$ from winning (such a strategy exists since $Y_{\bar{\sigma}} = \emptyset$). If the play visits a vertex $v \in N$, then Player σ will move to any successor v' inside X_σ. Such a successor must exist since vertices in N are never dead-ends in G. Furthermore, they belong to V_σ, thus since $G[X_\sigma]$ is a subgame w.r.t. σ (by Lemma 1.1), they remain non-dead-ends in $G[X_\sigma]$. If the play visits $\mathrm{Attr?}_\sigma(G[X_\sigma], N) \setminus N$, then Player σ will use his strategy to either cause Player $\bar{\sigma}$ to be inconsistent, or to attract the play

Algorithm 2 Vertices in which no win is possible: `ComputeNoWin`

13 **Function** ComputeNoWin(G, σ, n,$W_{\bar{\sigma}}$)
14 $\quad nowin := W_{\bar{\sigma}}$.
15 $\quad$ **repeat**
16 $\qquad nowin' := nowin$
17 $\qquad X_{\bar{\sigma}} := \text{Attr?}_{\bar{\sigma}}(G, nowin)$
18 $\qquad X_{\sigma} := V \setminus X_{\bar{\sigma}}$
19 $\qquad N := \{v \in X_{\sigma} \mid \chi(v) = n\}$
20 $\qquad Y := X_{\sigma} \setminus \text{Attr!}_{\sigma}(G[X_{\sigma}], N)$
21 $\qquad (Y_0, Y_1, Y_{tie}) := \text{SolveGame}(G[Y])$
22 $\qquad nowin := X_{\bar{\sigma}} \cup Y_{\bar{\sigma}} \cup Y_{tie}$
23 $\quad$ **until** $nowin' = nowin$
24 $\quad$ **return** $nowin$

in a finite number of steps to N or $D'_{\sigma} \cup D_{tie}$ (such a strategy exists by the definition of a may-attractor set). We use D'_{σ} to denote the dead-end vertices of Player σ in $G[X_{\sigma}]$. Since $G[X_{\sigma}]$ is not necessarily a subgame w.r.t. $\bar{\sigma}$, D'_{σ} may contain non-dead-end vertices of Player $\bar{\sigma}$ from G that became dead-ends in $G[X_{\sigma}]$. However, this means that all their successors are in $X_{\bar{\sigma}}$, and as stated before Player $\bar{\sigma}$ cannot move consistently from X_{σ} to $X_{\bar{\sigma}}$, thus he cannot win in them in G as well.

It is easy to see that this strategy indeed prevents Player $\bar{\sigma}$ from winning. □

Corollary 1. *The result of* `ComputeOpponentWin` *is the full winning region of Player $\bar{\sigma}$ in G.*

In the original algorithm in [23], given the set $W_{\bar{\sigma}}$, we could conclude that all the remaining vertices form the winning region of Player σ in G. Yet, this is not the case here. We now divide the remaining vertices into W_{tie} and W_{σ}. We first compute the set *nowin* of vertices in G from which Player σ does not have a winning strategy, i.e. Player $\bar{\sigma}$ has a strategy that prevents Player σ from winning. This is again done iteratively, by the function `ComputeNoWin`, given as Algorithm 2.

The algorithm `ComputeNoWin` resembles the algorithm `ComputeOpponentWin`. The initialization here is to $W_{\bar{\sigma}}$, since this is clearly a non-winning region of Player σ. Furthermore, in this case after the recursive call to `SolveGame`$(G[Y])$, the set $X_{\bar{\sigma}}$ is extended not only by the winning region of Player $\bar{\sigma}$ in $G[Y]$, $Y_{\bar{\sigma}}$, but also by the tie-region Y_{tie} (line 22). Apart from those differences, one can see that the only difference is that the use of a must-attractor set is replaced by a may-attractor set and vice versa. This is because in the case of `ComputeOpponentWin` we are after a definite win of Player $\bar{\sigma}$, whereas in the case of `ComputeNoWin` we also allow a tie, therefore may edges take a different role. Namely, in this case, when we extend the current set *nowin* (line 17), we use a may-attractor set of Player $\bar{\sigma}$ because when our goal is to prevent Player σ from winning, we allow Player $\bar{\sigma}$ to be inconsistent. On the other hand, in the computation of Y we now remove from $X_{\bar{\sigma}}$ only the vertices from which Player σ can *consistently* attract the play to the maximum priority (using the must-attractor set, line 20). This is because only such vertices cannot contribute to the goal of preventing Player σ from winning. Other vertices where he can reach the maximum priority, but only at the expense of consistency can still be of use for this goal.

Lemma 4. *1. For every X_σ as used in Algorithm 2, $G[X_\sigma]$ is a subgame.*
2. For every Y as used in Algorithm 2, $G[Y]$ is a subgame.

Again, the maximum priority in $G[Y]$ is smaller than n, which is why the recursion terminates.

Lemma 5. *At the beginning of each iteration, the set nowin is a non-winning region for Player σ in G.*

This lemma that can be shown with a careful analysis ensures that the final result *nowin* of `ComputeNoWin` is indeed a subset of the non-winning region of Player σ in G. It remains to show that no non-winning vertices are missing.

Lemma 6. *When $nowin' = nowin$, then $V \setminus nowin$ is a winning region for Player σ in G.*

Proof. When $nowin' = nowin$, it must be the case that the last iteration of `SolveGame` ended with $Y_{\bar{\sigma}} = Y_{tie} = \emptyset$, and $nowin = X_{\bar{\sigma}}$. Therefore it suffices to show that $V \setminus X_{\bar{\sigma}} = X_\sigma$ is a winning region for Player σ in G.

Clearly, Player $\bar{\sigma}$ cannot move from X_σ to $X_{\bar{\sigma}}$. Otherwise the vertex $v \in X_\sigma$ where this is done belongs to $\mathrm{Attr?}_{\bar{\sigma}}(G, X_{\bar{\sigma}})$ and therefore to $X_{\bar{\sigma}}$ as well. This contradicts $v \in X_\sigma$. Hence, Player $\bar{\sigma}$ is "trapped" in X_σ and as $G[X_\sigma]$ is a subgame, Player σ is never obliged to move to $X_{\bar{\sigma}}$.

Consider the case where the play stays in X_σ. In order to win, Player σ will play as follows. If the current configuration is in Y, then Player σ will use his winning strategy on $G[Y]$ (such a strategy exists since $Y_{\bar{\sigma}} = Y_{tie} = \emptyset$ and $Y_\sigma = Y$). If the play visits a vertex $v \in N$, then Player σ will move to a must successor v' inside X_σ. Such a successor exists because otherwise $v \in \mathrm{Attr?}_{\bar{\sigma}}(G, X_{\bar{\sigma}})$ and hence also in $X_{\bar{\sigma}}$, in contradiction to $v \in N \subseteq X_\sigma$. If the play visits $\mathrm{Attr!}_\sigma(G[X_\sigma], N) \setminus N$, then Player σ will attract it in a finite number of steps to N or D_σ, while being consistent.

This strategy ensures that Player σ is consistent and is indeed winning. $\square$

Corollary 2. `ComputeNoWin` *returns the full non-winning region of Player σ in G.*

We can now conclude that the remaining vertices in $V \setminus nowin$ form the full winning region of Player σ in G, and the tie region in G is exactly $nowin \setminus W_{\bar{\sigma}}$. This is the set of vertices from which neither player wins.

Solving the game is now achieved by Function `SolveGame` shown in Algorithm 3.

We have suggested an algorithm for computing the winning (and non-winning) regions of the players. The correctness proofs also show how to define strategies for the players. Yet, we omit this discussion due to space limitations. The algorithm can also be used for checking a concrete system in which all may-edges are also must-edges and $V_{tie} = \emptyset$.

Remark 1. Let G be a parity game in which $V_{tie} = \emptyset$ and all edges are must. Then W_{tie} computed by the algorithm `SolveGame` is empty.

Algorithm 3 The main function: `SolveGame`

Function SolveGame(G)
  $n := max\{\chi(v) \mid v \in V\}$
  if $n = 0$ **then** // *return* $(W_0,\ W_1,\ W_{tie})$
    return $(V \setminus \text{Attr?}_1(G, \emptyset),\ \text{Attr!}_1(G, \emptyset),\ \text{Attr?}_1(G, \emptyset) \setminus \text{Attr!}_1(G, \emptyset))$
  else
    $\sigma := n \mod 2$
    $W_{\bar{\sigma}} := \text{ComputeOpponentWin}(G,\ \sigma,\ n)$
    $W_{\sigma} := V \setminus \text{ComputeNoWin}(G,\ \sigma,\ n,\ W_{\bar{\sigma}})$
    $W_{tie} := V \setminus (W_{\bar{\sigma}} \cup W_{\sigma})$
    return $(W_0,\ W_1,\ W_{tie})$

Complexity Let l and m denote the number of vertices and edges of G. Let n be the maximum priority. A careful analysis shows that the algorithm is in $O((l+m)^{n+1})$.

Theorem 5. *Function* `SolveGame` *computes the winning regions* (W_0, W_1, W_{tie}) *for a given parity game in time exponential in the maximal priority. Additionally, it can be used to determine the winning strategy for the corresponding winner.*

We conclude that when applied to a model checking game $\Gamma_{\mathcal{T}}(s_0, \varphi_0)$, the complexity of `SolveGame` is exponential in the alternation depth of φ_0.

6 Refinement of Generalized Parity Games

Assume we are interested to know whether a concrete state s_c satisfies a given formula φ. Let (W_0, W_1, W_{tie}) be the result of the previous algorithm for the parity game obtained by the model checking game. Assume the vertex $v = s_a \vdash \varphi$, where s_a is the abstract state of s_c, is in W_0 or W_1. Then the answer is clear: $s_c \models \varphi$ if $v \in W_0$ and $s_c \not\models \varphi$ if $v \in W_1$. Otherwise, the answer is indefinite and we have to refine the abstraction to get the answer.

As in most cases, our refinement consists of two parts. First, we choose a criterion telling us how to split abstract states. We then construct the refined abstract model using the refined abstract state space. In this section we study the first part.

Given that $v \in W_{tie}$, our goal in the refinement is to find and eliminate at least one of the causes of the indefinite result. Thus, the criterion for splitting the abstract states is obtained from a *failure vertex*. This is a vertex $v' = s'_a \vdash \varphi'$ s.t. (1) $v' \in W_{tie}$; (2) the classification of v' to W_{tie} *affects* the indefinite result of v; and (3) the indefinite classification of v' can be *changed* by splitting it. The latter requirement means that v' *itself* is responsible for introducing (some) uncertainty. The others demand that this uncertainty is relevant to the result in v.

The game solving algorithm is adapted to remember for each vertex in W_{tie} a failure vertex, and a failure reason. We distinguish between the case where $n = 0$ and the case where $n \geq 1$ in `SolveGame`.

$\boldsymbol{n = 0}$: In this case the set W_{tie} is computed by $\mathrm{Attr?}_1(G, \emptyset) \setminus W_1$. Note that W_1 is already updated when the computation of $\mathrm{Attr?}_1(G, \emptyset)$ starts. We now enrich the computation of $\mathrm{Attr?}_1(G, \emptyset)$ to record failure information for vertices which are not in W_1 and thus will be in W_{tie}.

In the initialization we have two possibilities: (1) vertices in D_1, which are clearly not in W_{tie}, thus no additional information is needed; and (2) vertices in D_{tie}, for which the failure vertex and reason are the vertex itself [failDE].

As for the iteration, suppose we have $\mathrm{Attr?}_1^i(G, \emptyset)$, with the additional information attached to every vertex in it which is not in W_1. We now compute the set $\mathrm{Attr?}_1^{i+1}(G, \emptyset)$. Let v' be a vertex that is added to $\mathrm{Attr?}_1^{i+1}(G, \emptyset)$. If $v' \in W_1$, then no information is needed. Otherwise, we do the following.

1. If $v' \in V_1$ and there exists a may edge $v' \xrightarrow{may} v''$ s.t. $v'' \in W_1$, then v' is a failure state, with this edge being the reason [failP1].
2. If $v' \in V_0$ and has a may edge $v' \xrightarrow{may} v''$ s.t. $v'' \notin \mathrm{Attr?}_1^i(G, \emptyset)$, then v' is a failure state, with this edge being the reason [failP0].
3. Otherwise, there exists a may (that is possibly also a must) edge $v' \xrightarrow{may} v''$ s.t. $v'' \in \mathrm{Attr?}_1^i(G, \emptyset) \setminus W_1$. The failure state and reason of v' are those of v''.

Note that the order of the "if" statements in the algorithm determines the failure state returned by the algorithm. Different heuristics can be applied regarding their order. A careful analysis shows the following.

Lemma 7. *The computation of failure vertices for $n = 0$ is well defined, meaning that all the possible cases are handled. Furthermore, if the failure reason computed by it is a may edge, then this edge is* not *a must edge.*

Intuitively, during each iteration of the computation, if the vertex $v' \in W_{tie}$ that is added to $\mathrm{Attr?}_1^{i+1}(G, \emptyset)$ is not responsible for introducing the indefinite result (cases 1 and 2), then the computation greedily continues with a vertex in W_{tie} that *affects* its indefinite classification (case 3).

There are three possibilities where we say that the vertex itself is responsible for ? and consider it a failure vertex: failDE, failP1 and failP0. For a vertex in V_{tie} (case failDE), the failure reason is clear. Consider case failP1. In this case $v' \in V_1$ is considered a failure vertex, with the may edge to $v'' \in W_1$ being the failure reason. By Lemma 7 we have that it is *not* a must edge. The intuition for v' being a failure vertex is that if this edge was a must edge, it would change the classification of v' to W_1. If no such edge existed, then v' would not be added to $\mathrm{Attr?}_1^{i+1}(G, \emptyset)$ and thus to W_{tie}. Finally, consider case failP0. In this case $v' \in V_0$ has a may edge to v'' which is still unclassified at the time v' is added to $\mathrm{Attr?}_1(G, \emptyset)$. This edge is considered a failure reason because if it was a must edge rather than a may edge then v' would remain unclassified as well for at least one more iteration. Thus it would have a better chance to eventually remain outside the set $\mathrm{Attr?}_1^i(G, \emptyset)$ until the fixpoint is reached, changing the classification of v' to W_0.

***$n \geq 1$**:* In this case the set W_{tie} is computed by $V \setminus (W_{\bar{\sigma}} \cup W_{\sigma})$. This equals `ComputeNoWin`$(G, \sigma, n, W_{\bar{\sigma}}) \setminus W_{\bar{\sigma}}$, where $W_{\bar{\sigma}}$ is already updated when the computation of `ComputeNoWin`$(G, \sigma, n, W_{\bar{\sigma}})$ starts. Similarly to the previous case, we enrich the computation of `ComputeNoWin`$(G, \sigma, n, W_{\bar{\sigma}})$, and remember a failure vertex for each vertex which is not in $W_{\bar{\sigma}}$ and thus will be in W_{tie}.

In each iteration of `ComputeNoWin` the vertices added to the computed set are of three types: $X_{\bar{\sigma}}$, $Y_{\bar{\sigma}}$ and Y_{tie}.

The set $X_{\bar{\sigma}}$ is computed by $\mathrm{Attr?}_{\bar{\sigma}}(G, nowin)$. Thus in order to find failure vertices for such vertices that are not in $W_{\bar{\sigma}}$ we use an enriched computation of the may-attractor set, as described in the case of $n = 0$. This time the role of W_1 is replaced by $W_{\bar{\sigma}}$, 0 is replaced by σ and 1 by $\bar{\sigma}$. Furthermore, in the initialization of the computation we now also have the set *nowin* from the previous iteration, for which we already have the required information.

Vertices in Y_{tie} already have a failure vertex and reason, recorded during the computation of `SolveGame`$(G[Y])$.

We now explain how to handle vertices in $Y_{\bar{\sigma}}$. Such vertices have the property that Player $\bar{\sigma}$ wins from them in $G[Y]$. Hence, as long as the play stays in $G[Y]$, Player $\bar{\sigma}$ wins. Furthermore, Player $\bar{\sigma}$ can always stay in $G[Y]$ in his moves. Thus, for a vertex v' in $Y_{\bar{\sigma}}$ that is *not* in $W_{\bar{\sigma}}$ it must be the case that Player σ can force the play out of $G[Y]$ and into $(V \setminus Y) \setminus W_{\bar{\sigma}}$ (If the play reaches $W_{\bar{\sigma}}$ then Player $\bar{\sigma}$ can win after all). Thus, $v' \in \mathrm{Attr?}_{\sigma}(G, (V \setminus Y) \setminus W_{\bar{\sigma}})$. Let $\bar{Y} = V \setminus Y$ be the set of vertices outside $G[Y]$. We get that $Y_{\bar{\sigma}} \setminus W_{\bar{\sigma}} = Y_{\bar{\sigma}} \cap \mathrm{Attr?}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$. Therefore, to find the failure reason in such vertices, we compute $\mathrm{Attr?}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$. During this computation, for each vertex v' in $Y_{\bar{\sigma}}$ that is added to the attractor set (and thus will be in W_{tie}) we choose the failure vertex and reason based on the reason for v' being added to the set. This is because if the vertex was not in $\mathrm{Attr?}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$, it would be in $W_{\bar{\sigma}}$ in G as well. The information is recorded as follows.

In the initialization of the computation we have vertices in D_{σ}, D_{tie} or $\bar{Y} \setminus W_{\bar{\sigma}}$ which are clearly not in $Y_{\bar{\sigma}}$, thus no additional information is needed.

As for the iteration, suppose we have $\mathrm{Attr?}^{i}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$, with the additional information attached to every vertex in it which is in $Y_{\bar{\sigma}}$ (by the above equality such a vertex is not in $W_{\bar{\sigma}}$). We now compute the set $\mathrm{Attr?}^{i+1}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$. Let v' be a vertex that is added to $\mathrm{Attr?}^{i+1}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}})$. If $v' \notin Y_{\bar{\sigma}}$, then no information is needed. Otherwise, we do the following.

1. If $v' \in V_{\sigma}$ and there exists a may edge $v' \xrightarrow{may} v''$ which is *not* a must edge s.t. $v'' \in \bar{Y} \setminus W_{\bar{\sigma}}$, then v' is a failure state, with this edge being the reason.
2. If $v' \in V_{\sigma}$ and it has a must edge to $v'' \in X_{\bar{\sigma}} \setminus W_{\bar{\sigma}}$, then we set the failure vertex and reason of v' to be those of v'' (which are already computed).
3. Otherwise, v' has a may (possibly must) edge to a vertex $v'' \in \mathrm{Attr?}^{i}_{\sigma}(G, \bar{Y} \setminus W_{\bar{\sigma}}) \cap Y_{\bar{\sigma}}$. In this case the failure state and reason of v' are those of v''.

Lemma 8. *The computation of failure vertices for $n \geq 1$ is well defined, meaning that all the possible cases are handled.*

Intuitively, in case 1, v' is considered a failure state, with the may (not must) edge to $v'' \in \bar{Y} \setminus W_{\bar{\sigma}}$ being the reason because if this edge did not exist, v' would not be added to the may-attractor set, and thus would remain in $W_{\bar{\sigma}}$ in G. A careful analysis shows that the only possibility where there exists such a *must* edge to $v'' \in \bar{Y} \setminus W_{\bar{\sigma}}$ is when this edge is to $X_{\bar{\sigma}} \setminus W_{\bar{\sigma}}$. This is handled separately in case 2. The set $X_{\bar{\sigma}} \setminus W_{\bar{\sigma}}$ is a subset of W_{tie} for which the failure was already analyzed, and in case 2 we set the failure vertex and reason of v' to be those of $v'' \in X_{\bar{\sigma}} \setminus W_{\bar{\sigma}}$. This is because changing the classification of v'' to $W_{\bar{\sigma}}$ would make a step in the direction of changing the classification of $v' \in V_\sigma$ to $W_{\bar{\sigma}}$ as well. Similarly, since the edge from v' to v'' is a must edge, changing the classification of v'' to W_σ would change the classification of $v' \in V_\sigma$ to W_σ. In all other cases, the computation recursively continues with a vertex in $Y_{\bar{\sigma}}$ that was already added to the may-attractor set and that *affects* the addition of v' to it (case 3).

This concludes the description of how `SolveGame` records the failure information for each vertex in W_{tie}. A simple case analysis shows the following.

Theorem 6. *Let v_f be a vertex that is classified by* `SolveGame` *as a failure vertex. The failure reason can either be the fact that $v_f \in V_{tie}$, or it can be an edge $(v_f, v') \in \xrightarrow{may} \setminus \xrightarrow{must}$.*

Once we are given a failure vertex $v' = s'_a \vdash \varphi'$ and a corresponding reason for failure, we guide the refinement to discard the cause for failure in the hope for changing the model checking result to a definite one. This is done as in [18], where the failure information is used to determine how the set of concrete states represented by s'_a should be split in order to eliminate the failure reason. A criterion for splitting *all* abstract states can then be found by known techniques, depending on the abstraction used (e.g. [4, 2]).

After refinement, one has to re-run the model checking algorithm on the game graph based on the refined KMTS to get a definite value for s_c and φ. However, we can restrict this process to the previous W_{tie}. When constructing the game graph based on the refined KMTS, every vertex $s_a^2 \vdash \varphi'$ for which a vertex $s_a \vdash \varphi'$ (where s_a^2 results from splitting s_a) exists in W_0 or W_1 in the previous game graph can be considered a dead end winning for Player 0 or Player 1, respectively. In this way we avoid unnecessary refinement.

7 Conclusion

This work presents a game-based model checking for abstract models with respect to specifications in μ-calculus, interpreted over a 3-valued semantics, together with automatic refinement, if the model checking result is indefinite.

The closest work to ours is [18], in which a game-based framework is suggested for abstraction-refinement for CTL with respect to a 3-valued semantics. While it is relatively simple to extend their approach to alternation-free μ-calculus, the extension to full μ-calculus is not trivial. This is because, in the game graph for alternation-free μ-calculus each strongly connected component can be uniquely identified by a single fixpoint. For full μ-calculus, this is not the case any more, thus a more complicated algorithm is needed in order to determine who has the winning strategy.

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
2. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, *LNCS* 1855, 2000.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, Dec. 1999.
4. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine leraning techniques. In *CAV*, 2002.
5. R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Inf.*, 27:725–747, 1990.
6. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2), March 1997.
7. E. A. Emerson, and C. S. Jutla Tree automata, μ-calculus and determinacy. In *Proc. 32th Symp. on Foundations of Computer Science (FOCS'91)*, pp. 368–377, 1991, *IEEE Computer Society Press.*
8. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Logic in Computer Science (LICS)*, 1986.
9. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Computer-Aided Verification (CAV)*, *LNCS* 2404, pp. 137–150, 2002.
10. P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *VMCAI*, *LNCS* 2575, pp. 206–222, 2003.
11. M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *European Symposium on Programming (ESOP)*, LNCS 2028, pp. 155–169, 2001.
12. D. Kozen. Results on the Propositional μ-calculus. *TCS*, 27: 333-354, 1983.
13. W. Lee, A. Pardo, J.-Y. Jang, G. D. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD*, pp. 76–81, 1996.
14. 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.
15. D. Long, A. Browne, E. Clark, S. Jha, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV*, *LNCS* 818, pp. 338–350, 1994.
16. A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional mu-calculus model checking. In *Computer Aided Verification (CAV)*, 1997.
17. A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference (DAC)*, pp. 457–462, 1998.
18. S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinemnet. In *Computer Aided Verification (CAV)*, *LNCS* 2725, pp. 275–287, 2003.
19. C. Stirling. Local model checking games. In *Concurrency Theory (CONCUR)*, *LNCS* 962, pp. 1–11, 1995.
20. C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. In *Theory and Practice of Software Development*, LNCS, 1989.
21. A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math.*, 5:285–309, 1955.
22. G. Winskel. Model checking in the modal ν-calculus. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1989.
23. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.