

Local abstraction-refinement for the mu-calculus^{*}

Harald Fecher¹ and Sharon Shoham²

¹ Christian-Albrechts-University Kiel, Germany
hf@informatik.uni-kiel.de

² The Technion, Haifa, Israel
sharonsh@cs.technion.ac.il

Abstract. Counterexample-guided abstraction refinement (CEGAR) is a key technique for the verification of computer programs. Grumberg et al. developed a CEGAR-based algorithm for the modal μ -calculus. There, every abstract state is split in a refinement step. In this paper, the work of Grumberg et al. is generalized by presenting a new CEGAR-based algorithm for the μ -calculus. It is based on a more expressive abstract model and applies refinement only locally (at a single abstract state), i.e., the *lazy abstraction* technique for safety properties is adapted to the μ -calculus. Furthermore, it separates refinement determination from the (3-valued based) model checking. Three different heuristics for refinement determination are presented and illustrated.

1 Introduction

One of the most successful techniques to checking correctness of large or even infinite programs is predicate abstraction [8] with *counterexample-guided abstraction refinement* (CEGAR) [3]. This approach consists of three phases: abstraction, model checking, and refinement. A typical tool based on that technique is SLAM [2], where an efficient approximation of the post-transitions of a concrete system is calculated by using cartesian approximation, and where a spurious counterexample found during the model checking phase is used for determining the refinement. Another prominent tool based on CEGAR is BLAST [14], where, contrary to SLAM, refinement is applied locally (called *lazy abstraction*), i.e., only the relevant abstract states of a trace being a spurious counterexample are refined. Both tools mentioned are only capable of verifying safety properties.

Grumberg et al. [9, 10] present CEGAR-based algorithms for the verification of the μ -calculus [19], which is a powerful formalism for expressing branching time³ and reachability properties by using fixpoint constructions. These approaches have as underlying abstract models *Kripke modal transition systems* [15], which have may and must transitions (over, resp., under approximation of the concrete transitions), as in *modal transition systems* [20]. Two transition relations are essential in order to preserve branching time properties. They also allow to preserve both *validity* and *invalidity* from the abstract model to the concrete model, at the cost of introducing a third truth value

^{*} This work is in part financially supported by the DFG project *Refism* (FE 942/1-1)

³ Branching time is relevant whenever nondeterminism occurs from external factors (e.g., user input), from random behavior, or from the modeling of faulty systems or channels.

unknown, which means that the truth value in the concrete model is unknown. This leads to a *3-valued semantics*. In this setting, refinement is no longer needed when the result is *invalid*, as in traditional CEGAR approaches. Instead, refinement is needed when the result is *unknown*. As such, the role of a counterexample as guiding the refinement is taken by some cause of the indefinite result.

In [9], a 3-valued satisfaction game is defined, where the Verifier tries to obtain validity, and the Falsifier tries to obtain invalidity. In order to win, a player must not use may transitions. The third truth value is captured by the possibility that none of the players wins. Furthermore, their model checking algorithm, which is a generalization of the parity game algorithm of Zielonka [28], determines an abstract state z and a predicate p such that the splitting of z with respect to p leads to less spurious behavior. This approach is generalized in [10] by making the approach independent from the Zielonka algorithm, i.e., allowing more efficient algorithms [18]. There, the model checking is performed via a reduction of the 3-valued satisfaction game into two games: one for validity and one for invalidity. The predicate determining the splitting is derived from the trace obtained after playing the *non-losing* strategies of the players in these games against each other. In both approaches, every configuration (abstract states combined with subproperties) where the (in)validity is not yet shown is split, i.e., only a weak form of *lazy abstraction* is made.

Contribution. A new CEGAR-based model checking algorithm for the μ -calculus is presented. This algorithm improves the approaches of [9, 10] in the following way:

- A more expressive underlying abstract model is used, namely *generalized Kripke modal transition systems* [26], where must hypertransitions, as in *disjunctive modal transition systems* [21], are used, i.e., a must transition points to a set of states rather than to a singleton. Consequently, a smoother refinement determination can be obtained [26] and more properties can in principle be shown [6].
- A stronger notion of *lazy abstraction* is used: only a single abstract state is split. Even better, some but not all configurations having the same underlying abstract state are split. Thus the state space remains smaller and verification is sped up.
- The algorithm provides a separation of the refinement determination from the model checking. This is done by providing a structure that encodes all possible causes for the indefinite result. On this structure, heuristics for determining the local refinement step can be defined. In particular, three different heuristics are presented and illustrated. The most promising one can only be defined in a local refinement setting.

Further related work. A CEGAR-approach to branching time properties is given in [24], where, contrary to our approach, only the transition relation is under, resp., over approximated (the state space remains equal). In [12], the techniques used in SLAM are generalized to branching time properties, where the underlying abstract model is equivalent to Kripke modal transition systems. In [23] models are abstracted by *alternating transitions systems* with *focus predicates*. These resemble game-graphs with must hypertransitions. Refinement is not discussed in this paper. A CEGAR-approach for the more general alternating μ -calculus is given in [1], which is a generalization of [5]. In [1] the underlying abstract model has must as well as may hypertransitions. Refinement is made globally (not locally) and the refinement determination depends on the

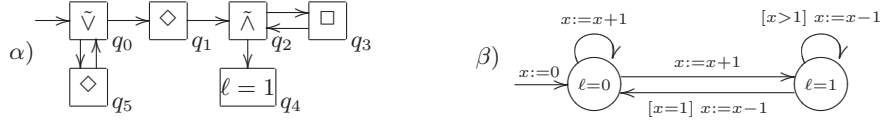


Fig. 1. A μ -calculus formula (α) in terms of automata (see Section 3.3), and a system (β). α): The property at the initial state q_0 holds if (i) there is a transition such that $\ell = 1$ holds on every possible path or (ii) there is a transition such that q_0 holds again (consequently, if there is an infinite path then q_0 holds). β): The range of ℓ is $\{0, 1\}$ and of x is \mathbb{N} , both initialized with 0. The actions of the transitions can be executed, including the modification of ℓ , whenever the guard, depicted in rectangular brackets, is valid. When the guard is *true*, it is simply omitted.

model checking algorithm, i.e., no separation is used. Must and may hypertransitions are also used in [7], where finite-state abstractions can be computed (for any μ -calculus formula) by a generalization of predicate abstraction. No CEGAR-based algorithm is presented there. In [25] a different kind of may hypertransitions is used in order to improve precision for non-partitioning abstraction functions. Our approach does not need these may hypertransitions for precision, since our abstraction function locally corresponds to partitions. [25] also suggests a CEGAR-based algorithm, however they consider only the alternation-free fragment of the μ -calculus. Moreover, their refinement follows [9], resulting in a weak form of lazy abstraction. In [11] the techniques of testing and verification interact with each other, improving the refinement heuristic. Similar improvements can be obtained by using 3-valued abstract models, which we do.

Outline. The new CEGAR-based algorithm is illustrated by an example in Section 2, made precise in Section 4, and is improved in Section 6. Section 3 presents the underlying concrete/abstract models, game structures, and the μ -calculus in terms of alternating tree automata. The heuristics for refinement determination are developed in Section 5 and Section 7 concludes the paper. An appendix contains pseudo codes of less important procedures. Proofs are omitted due to space constraints.

2 Example

Our model checking algorithm is illustrated by checking the μ -calculus formula, presented via a tree automaton description in Figure 1 (α), at the system depicted in Figure 1 (β). Note that both the formula and the system are used for illustration purposes and do not claim practical relevance.

The model checking is based on a configuration structure, where each configuration consists of a subproperty and a (possibly abstract) state of the system. The outgoing transitions of a configuration define ‘subgoals’ for determining the value (valid, invalid or unknown) of the subproperty in the (abstract) state of the system. Subproperties are given by the automaton states, which are labeled either by a predicate (e.g. $\ell = 1$) or by $\tilde{\wedge}$, $\tilde{\vee}$, \diamond , \square . Intuitively, $\tilde{\wedge}$ and $\tilde{\vee}$ stand for the logical connectives \wedge and \vee resp. Similarly, \diamond stands for “exists a successor”, while \square stands for “all successors”.

The first configuration structure is obtained by combining all subproperties (automaton states) with the single abstract-state *true*, which abstracts any concrete system-

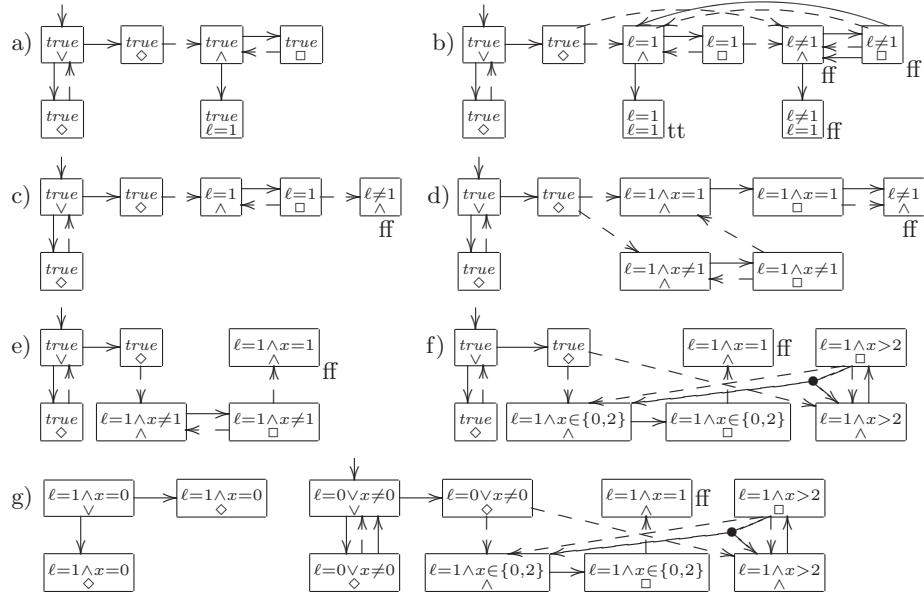


Fig. 2. Example of a property check via local refinement. May transitions are depicted as dashed arrows and must, as well as junction, transitions as solid arrows

state. In addition, the transition relation of the system is overapproximated by a may transition from $true$ to itself. No must transition (underapproximation) is used in the initial abstraction. The obtained configuration structure is presented in Figure 2 (a). For readability, the figure uses the labels of the automaton-states, rather than their names. May and must transitions leave \diamond - or \square -configurations and the other transitions, which imitate the automaton transitions, are called *junction transitions*.

In general, the algorithm iterates four phases: (in)validity determination, simplification of the configuration structure, refinement determination by some heuristic, and local refinement. The validity of the configurations is determined via a parity game algorithm, where the Verifier moves in \checkmark - and \diamond -configurations, and the Falsifier moves in $\bar{\wedge}$ - and \square -configurations. In the validity game the Verifier can only use must and junction transitions, whereas the Falsifier can additionally use may transitions. The valid configurations become labeled with tt . Thereafter, the same is done via an invalidity check where the Falsifier can only use must and junction transitions, whereas the Verifier can additionally use may transitions. The invalid configurations become labeled with ff . No validity or invalidity can be determined in (a). As a result no simplification is possible in this case. The unknown values in (a) result from four possible causes. One is the configuration $(true, \ell = 1)$, where the validity of the predicate $\ell = 1$ in the state $true$ is unknown, thus neither the Verifier nor the Falsifier can win. The others are the three may transitions in the configuration structure, which result from the may transition from the $true$ state to itself. For example, the fact that the may transition from $(true, \diamond)$ to $(true, \checkmark)$ is not a must transition prevents the Verifier from winning the validity game, and on the other hand, its existence interferes with the winning of

the Falsifier in the invalidity game. These causes represent all the possible causes for an indefinite result. Consequently, in order to refine the system, a heuristic determines either (i) a configuration where the property is a predicate and the validity is unknown or (ii) a may transition for which no corresponding must transition exists.

Assuming the heuristic yields the configuration $(true, \ell = 1)$, whose validity is unknown, then all configurations forwardly/backwardly reachable from $(true, \ell = 1)$ via junction transitions are split by the predicate $\ell = 1$ during the local refinement phase. The may and must (hyper)transitions incoming and leaving the new configurations are recalculated via suitable satisfiability checks solved by a theorem prover. As in [26], a may transition from an abstract state z_1 to another z_2 exists iff there is a transition from a concrete state abstracted by z_1 to a concrete state abstracted by z_2 . A must (hyper)transition from z_1 to a set of abstract states \tilde{Z} exists iff every concrete state abstracted by z_1 has a transition with a target that is abstracted by an element from \tilde{Z} . Thereby, (b) is obtained. Note that we do not split the state $true$ in all the configurations. Instead, it is split only in the configurations forwardly/backwardly reachable from $(true, \ell = 1)$ via junction transitions. This makes our abstraction *lazy*.

The next iteration starts from (b). The tt and ff labels describe the result after making the (in)validity-determinations as described before. Unlike the initial configuration structure, in this case, some of the configurations are determined as (in)valid. Thereafter, configurations and transitions having no further influence on the (in)validity-determinations, are removed in the simplification phase, yielding (c). For example, the junction transition from $(\ell = 1, \wedge)$ to $(\ell = 1, \ell = 1)$ along with the target configuration, which is labeled tt, are removed, since knowing that one conjunct has value tt, makes the value of \wedge depend on the value of the other conjunct. The algorithm continues with the simplified structure. Assuming the heuristic determines the may transition pointing to $(\ell \neq 1, \wedge)$, then the source (and all configurations connected to it via junction transitions) are split by the weakest precondition to reach $\ell \neq 1$ in the concrete system, which is $\ell = 0 \vee x = 1$. Thus we obtain (d). Proceeding with (in)validity-determinations and simplifications, we obtain (e). Assuming the heuristic yields the may transition into $(\ell = 1 \wedge x \neq 1, \wedge)$, then the source (and all configurations connected to it via junction transitions) are split by the weakest precondition to reach $\ell = 1 \wedge x \neq 1$ in the concrete system, which is $(\ell = 0 \wedge x \neq 0) \vee (\ell = 1 \wedge x > 2)$. Thus we obtain (f), where a must hypertransition arises. No further validity or invalidity can be determined in (f), thus no simplification takes place. Assuming the heuristic yields the may transition into $(true, \vee)$, then the source (and all configurations connected to it via junction transitions) are split by the weakest precondition to reach $true$, which is $\ell = 0 \vee x \neq 0$. Thus we obtain (g), where the initial configuration is also recalculated. Now the initial configuration becomes valid and thus the property is verified.

3 Preliminaries

Throughout, $\mathbb{P}(B)$ denotes the power set of a set B . Functional composition is denoted by \circ . Given a relation $\rho \subseteq B \times D$ with subsets $X \subseteq B$ and $Y \subseteq D$ we write $X.\rho$ for $\{d \in D \mid \exists b \in X: (b, d) \in \rho\}$ and $\rho.Y$ for $\{b \in B \mid \exists d \in Y: (b, d) \in \rho\}$.

The projection to the i -th coordinate is denoted by π_i . Let $\text{map}(f, \Phi)$ be the sequence obtained from the sequence Φ by applying function f to all elements of Φ pointwise.

3.1 System

Without loss of generality, we will not consider action labels on models in this paper. A *rooted transition system* $T = (S, s^i, \rightarrow, \mathcal{L})$ consists of a (possibly infinite) set S of states, an initial state $s^i \in S$, a transition relation $\rightarrow \subseteq S \times S$, and a *predicate language* \mathcal{L} , which is a set of predicates that are interpreted over the states in S (i.e., each predicate $p \in \mathcal{L}$ denotes a set $\llbracket p \rrbracket \subseteq S$ of states), such that the following three conditions are satisfied. (i) There exists $p^i \in \mathcal{L}$ with $\llbracket p^i \rrbracket = \{s^i\}$. (ii) The boolean closure of \mathcal{L} , denoted by $\overline{\mathcal{L}}$, is a decidable theory (i.e., satisfiability is decidable). (iii) $\overline{\mathcal{L}}$ is effectively closed under exact predecessor operations; that is, for every formula ψ in $\overline{\mathcal{L}}$ we can compute the boolean combination $\text{pre}(\psi)$ of predicates from \mathcal{L} such that $\llbracket \text{pre}(\psi) \rrbracket = \rightarrow . \llbracket \psi \rrbracket$. In the following we assume a fixed rooted transition system $T = (S, s^i, \rightarrow, \mathcal{L})$.

3.2 Strong-weak-parity-game

Here, three valued parity games having under/over approximated transitions are presented. These games will be used to encode the satisfaction of a property in a system. They generalize the three-valued parity games of [10] by adding a validity function.

Definition 1. A strong-weak-parity-game $G = (C, C_1, C_2, c^i, R^-, R^+, \theta, \omega)$ has

- a set of game states C divided (not necessarily completely) by two players; $C_1 \subseteq C$ for Player 1 and $C_2 \subseteq C \setminus C_1$ for Player 2,
- an initial game state $c^i \in C$,
- a set of strong and a set of weak game transitions $R^-, R^+ \subseteq C \times C$,
- a parity function $\theta : C \rightarrow \mathbb{N}$ with finite image, and
- a validity function $\omega : C \rightarrow \{\text{tt}, \text{ff}, \perp\}$, into the values true, false, and unknown.

The source (target) of a transition t in G is denoted by $\text{src}(t)$, resp. $\text{tar}(t)$.

Definition 2. – Finite validity plays for strong-weak-parity-game G have the rules and winning conditions as stated in Table 1. An infinite play Φ is a win for Player 1 iff $\text{sup}(\text{map}(\theta, \Phi))$ is even; otherwise it is won by Player 2.

- Finite invalidity plays for G have the rules and winning conditions as stated in Table 2. An infinite play Φ is a win for Player 2 iff $\text{sup}(\text{map}(\theta, \Phi))$ is odd; otherwise it is won by Player 1.
- G is valid (is invalid) in $c \in C$ iff Player 1 (resp. Player 2) has a strategy for the corresponding validity (resp. invalidity) game such that Player 1 (resp. Player 2) wins all validity (resp. invalidity) plays started at c with her strategy. G is valid (is invalid) iff G is valid (resp. is invalid) in c^i .

Remark 1. The validity, as well as the invalidity, game obviously corresponds to a parity game. Therefore, decidability of validity, resp. invalidity, is in $\text{UP} \cap \text{coUP}$ [17].

$\omega(c) \neq \perp \vee c \notin C_1 \cup C_2$: Player 1 wins iff $\omega(c) = \text{tt}$
 $c \in C_1 \wedge \omega(c) = \perp$: Player 1 picks as next configuration $c' \in \{c\}.R^-$;
 $c \in C_2 \wedge \omega(c) = \perp$: Player 2 picks as next configuration $c' \in \{c\}.(R^- \cup R^+)$;

Table 1. Moves of validity game at game state c , specified through a case analysis. If a Player is unable to move at his turn, the other Player wins. Validity plays are sequences of game-states generated thus

$\omega(c) \neq \perp \vee c \notin C_1 \cup C_2$: Player 2 wins iff $\omega(c) = \text{ff}$
 $c \in C_1 \wedge \omega(c) = \perp$: Player 1 picks as next configuration $c' \in \{c\}.(R^- \cup R^+)$;
 $c \in C_2 \wedge \omega(c) = \perp$: Player 2 picks as next configuration $c' \in \{c\}.R^-$;

Table 2. Moves of invalidity game at game state c , specified through a case analysis. If a Player is unable to move at his turn, the other Player wins. Invalidity plays are sequences of game-states generated thus

Proposition 1. *Validation over strong-weak-parity-game is 3-valued, i.e., a strong-weak-parity-game is either valid, invalid, or neither of them.*

Definition 3. *A strong-weak-parity-game G is simplified if (i) it is valid or invalid in $c \in C$ iff $\omega(c) \neq \perp$ and (ii) there are no transitions (a) leaving (in)valid game-states, (b) leaving game-states from C_1 and point to invalid ones, or (c) leaving game-states from C_2 and point to valid ones, i.e., $\forall t \in R^+ \cup R^- : \omega(\text{sor}(t)) = \perp \wedge (\text{sor}(t) \in C_1 \Rightarrow \omega(\text{tar}(t)) \neq \text{ff}) \wedge (\text{sor}(t) \in C_2 \Rightarrow \omega(\text{tar}(t)) \neq \text{tt})$.*

Intuitively, G is simplified if the validity function encodes correctly all the (in)valid game-states, and in addition, only transitions that “explain” an unknown value exist.

Theorem 1. *For any strong-weak-parity-game G there is an equivalent simplified one G' in the sense that $C = C'$ and for all $c \in C$ we have: G is valid (is invalid) in c iff G' is valid (resp. is invalid) in c . Moreover, the algorithm from Table 3 calculates a corresponding G' .*

3.3 Property language

We will present the modal μ -calculus [19] in its equivalent form of automata [27].

Definition 4 (Tree automata). *An alternating tree automaton $A = (Q, q^i, \delta, \Theta)$ has*

- *a finite, nonempty set of states $(q \in)Q$ with the initial element $q^i \in Q$*
- *a transition relation δ mapping automaton states to one of the following forms, where q, q_1, q_2 are automaton states and $p \in \mathcal{L}$: $p \mid q \mid q_1 \tilde{\wedge} q_2 \mid q_1 \tilde{\vee} q_2 \mid \diamond q \mid \square q$*
- *an acceptance condition $\Theta: Q \rightarrow \mathbb{N}$ with finite image.*

An alternating tree automaton is depicted in Figure 1 (α), where all automaton-states have acceptance value 0. The labels of the automaton states and their outgoing transitions encode the transition relation δ . In the following, we assume a fixed alternating tree automaton $A = (Q, q^i, \delta, \Theta)$. Set Q_{qua} consists of those automaton-states of the

Algorithm Simplify (G : a strong-weak-parity-game)

- 1: Use a parity-game algorithm to determine the valid game-states and adapt ω accordingly.
- 2: Use a parity-game algorithm to determine the invalid game-states and adapt ω accordingly.
- 3: Remove in G all weak/strong transitions that (i) leave (in)valid game-state, (ii) leave elements from C_1 and point to invalid game states, or (iii) leave elements from C_2 and point to valid game states.

Table 3. Algorithm for the determination of equivalent, simplified strong-weak-parity-games, where $G = (C, C_1, C_2, c^i, R^-, R^+, \theta, \omega)$

form \diamond or \square , i.e., $Q_{\text{qua}} = \{q \in Q \mid \exists q' : \delta(q) \in \{\diamond q', \square q'\}\}$. The successor state of $q \in Q_{\text{qua}}$ is denoted by $\text{succ}(q)$, i.e., $\text{succ}(q) = q'$ if $\delta(q) \in \{\diamond q', \square q'\}$. Furthermore, $Q_1 = \{q \in Q \mid \delta(q) \in \bigcup_{q_1, q_2 \in Q} \{q_1, q_1 \tilde{\vee} q_2, \diamond q_1\}\}$ denotes the automaton-states under control of Player 1 and $Q_2 = \{q \in Q \mid \delta(q) \in \bigcup_{q_1, q_2 \in Q} \{q_1 \tilde{\wedge} q_2, \square q_1\}\}$ those under control of Player 2. Satisfaction of a rooted transition system with respect to an alternating tree automata is obtained via transformation into a strong-weak-parity-game:

Definition 5. The property-game for T and A , denoted $P_{T,A}$, is the strong-weak-parity-game $(S \times Q, S \times Q_1, S \times Q_2, (s^i, q^i), R^-, \{\}, \Theta \circ \pi_2, \omega)$, where

$$R^- = \{((s, q), (s, q')) \mid \exists q'' : \delta(q) \in \{q', q' \tilde{\wedge} q'', q'' \tilde{\wedge} q', q' \tilde{\vee} q'', q'' \tilde{\vee} q'\}\} \cup \{((s, q), (s', q')) \mid \delta(q) \in \{\diamond q', \square q'\} \wedge (s, s') \in \rightarrow\}$$

$$\omega(s, q) = \begin{cases} \text{tt} & \text{if } \delta(q) \in \mathcal{L} \wedge s \in \llbracket \delta(q) \rrbracket \\ \text{ff} & \text{if } \delta(q) \in \mathcal{L} \wedge s \notin \llbracket \delta(q) \rrbracket \\ \perp & \text{otherwise.} \end{cases}$$

Furthermore, we write $T \models q$, whenever $P_{T,A}$ is valid, and otherwise, we write $T \not\models q$ (which is equivalent to $P_{T,A}$ is invalid).

All the transitions in $P_{T,A}$ are strong. The transitions that leave game-states whose automaton component q is in Q_{qua} correspond to the transitions in the underlying system. In all other cases, the transitions reflect the automaton transitions, and the system component remains unchanged. The parity conditions also reflect the acceptance conditions of the automaton. ω evaluates game-states whose automaton component q is such that $\delta(q) \in \mathcal{L}$. In this case, the evaluation is determined by the value of the predicate $\delta(q)$ in s . The (in)validity of such game-states provides the basis of the (in)validity evaluation of the game. Note that our definition of $T \models q$ coincides with the standard definition of satisfaction, and $T \not\models q$ coincides with the satisfaction of the dual formula, i.e., corresponds to negation.

Next, special strong-weak-parity-games derived for alternating tree automata satisfaction on abstracted systems, in terms of generalized Kripke modal transition systems [26], are introduced. These are called *abstract property-games*. Unlike previous works, we do not define the abstract system separately. Instead, its description is intertwined with the property in the game structure. This is most convenient to enable a *lazy abstraction* where the same part of the system can be abstracted differently in different

contexts. The abstract property-games are obtained by combining the abstract-states $z \in Z$ with the property-states and encoding hypertransitions via additional game-states (hyper-points) where subsets of abstract-states $\tilde{Z} \in \mathbb{P}(Z)$ are combined with Q_{qua} . The hyper-points are used to model hypertransitions. The classification of game-states to players is based on the property-states as before, except that in hyper-points the responsibility of the players switches. Furthermore, an abstract state z has a formula describing the concrete states that are abstracted by z . In particular, the same concrete state can be abstracted by multiple abstract states. However, it will only be abstracted by a single abstract state in each context (property-state). Formally:

Definition 6. An abstract property-game P is a tuple (Z, ϱ, G) , where Z is a set of abstract states, $\varrho : Z \rightarrow \bar{\mathcal{L}}$ is an abstraction function, and G is a strong-weak-parity-game such that

- $C \subseteq (Z \times Q) \cup (\mathbb{P}(Z) \times Q_{\text{qua}})$,
- $C_i = C \cap ((Z \times Q_i) \cup (\mathbb{P}(Z) \times (Q_{\text{qua}} \setminus Q_i)))$ for $i \in \{1, 2\}$,
- an element $(\tilde{Z}, q) \in \mathbb{P}(Z) \times Q_{\text{qua}}$ encodes an hyper-point connecting $(z, q) \in Z \times Q_{\text{qua}}$ to (a subset of) the elements of \tilde{Z} combined with the next automaton state, $\text{succ}(q)$, i.e., $\forall (\tilde{Z}, q) \in C \cap (\mathbb{P}(Z) \times Q_{\text{qua}}) : R^- \cdot \{(\tilde{Z}, q)\} \subseteq \{(z, q) \mid z \in Z\}$ and $\{(\tilde{Z}, q)\} \cdot R^- \subseteq \{(z', \text{succ}(q)) \mid z' \in \tilde{Z}\}$.

P is simplified if G is.

To simplify the presentation of the paper, we refrain from formalizing the additional constraints of an abstract property-game. Instead, we describe them informally. Similarly to the property-game, the abstract property-game maintains the structure of the property automaton. In particular, whenever the automaton component is not in Q_{qua} , the outgoing game transitions are strong-transitions that reflect the automaton transitions, thus the system component does not change. When the automaton component is in Q_{qua} , the outgoing transitions reflect the transitions of the underlying system, except that they can now either overapproximate the system transitions, via weak-transitions, or underapproximate the system transitions, via strong-transitions that point to hyper-points. In analogy to generalized Kripke modal transition system, the weak transitions of an abstract property-game are also called *may transitions*, since they are used to represent may transitions of the underlying abstract model. The strong transitions of an abstract property-game that point to hyper-points are called *must transitions* (they represent must hypertransitions of the underlying model) and the other strong transitions are called *junction transitions*.

Recall that the may and must transitions leave game-states whose automaton state q is in Q_{qua} . In principle, if *some* concrete state abstracted by z has a transition to *some* concrete state abstracted by z' , i.e. $\varrho(z) \wedge \text{pre}(\varrho(z'))$ is satisfiable, then there exists a may transition from (z, q) to $(z', \text{succ}(q))$. This is called the *may condition*. A must transition from (z, q) to the hyper-point (\tilde{Z}, q) exists *only if* the *must condition* holds, namely *every* concrete state abstracted by z has a transition whose target state is abstracted by *some* state in \tilde{Z} , i.e. the implication $\varrho(z) \Rightarrow \text{pre}(\bigvee_{z' \in \tilde{Z}} \varrho(z'))$ holds. The hyper-point (\tilde{Z}, q) is connected via junction transitions to the game-states in $\{(z', \text{succ}(q)) \mid z' \in \tilde{Z}\}$. However, simplification can damage these rules. Including

additional may transitions that do not fulfill the may condition, or not including some of the must transitions although they do fulfill the must condition, is sound. However, a smaller set of may transitions, resp. a bigger set of must transitions makes the over, resp. under, approximation tighter and hence more precise. Similarly, the smaller the set Z in a hyper-point is, the more precise the must transition is.

The validity function ω is used as in the concrete property-game, except that now the evaluation of the predicate $p = \delta(q) \in \mathcal{L}$ in an abstract state z depends on the value of the predicate in *all* the concrete states abstracted by z . Namely, $\omega(z, q) = \text{tt}$, resp. ff , if $\varrho(z) \Rightarrow p$, resp. $\varrho(z) \Rightarrow \neg p$, holds. Otherwise, $\omega(z, q) = \perp$. The parity function is defined as in the concrete property-game (since it only depends on the automaton).

The initial abstraction for T , which contains only a single abstract state z abstracting everything (i.e., $\varrho(z) = \text{true}$), corresponds to the following abstract property-game:

Definition 7. *The initial abstract property-game $P_{T,A}^I$ for T and A is $(\{z\}, \{(z, \text{true})\}, (\{z\} \times Q, \{z\} \times Q_1, \{z\} \times Q_2, (z, q^i), R^-, R^+, \Theta \circ \pi_2, \omega))$, where z is an arbitrary element and*

$$\begin{aligned} R^- &= \{((z, q), (z, q')) \mid \exists q'' : \delta(q) \in \{q', q' \tilde{\wedge} q'', q'' \tilde{\wedge} q', q' \tilde{\vee} q'', q'' \tilde{\vee} q'\}\} \\ R^+ &= \{((z, q), (z, q')) \mid \delta(q) \in \{\diamond q', \square q'\}\} \\ \omega(z, q) &= \perp \quad \text{for } q \in Q. \end{aligned}$$

Note that the initial abstract property-game does not depend on T . This reflects the fact that we start with a fully abstracted system. In particular, no must transitions exist, and the may transitions correspond to a may transition from z to z in the underlying abstract system. The validity function interprets all the predicates as \perp in z .

Examples of abstract property-games for the system from Figure 1 (β) and the tree automaton from Figure 1 (α) appear in Figure 2. In particular, Figure 2 (a) presents the initial abstract property-game. In the figure, a game-state $(z, q) \in Z \times Q$ is labeled by $\varrho(z)$, which is the predicate describing the concrete states abstracted by z , and by the label of the automaton-state q , which reflects $\delta(q)$. To simplify the figure, hyper-points are omitted. Namely, instead of including a must transition from (z, q) to the hyper-point (\tilde{Z}, q) and junction transitions from the hyper-point to $\{(z', \text{succ}(q)) \mid z' \in \tilde{Z}\}$, Figure 2 directly connects (z, q) to $\{(z', \text{succ}(q)) \mid z' \in \tilde{Z}\}$ using a must transition, or, if necessary, a hypertransition.

4 CEGAR locally applied on configurations

In the verification algorithm, a simplified abstract property-game is calculated, starting from the initial abstract property-game. If the validity of the initial game-state remains unknown, a *refinement heuristic* is applied on the simplified abstract property-game.

Definition 8. *A refinement heuristic is a function mapping an abstract property-game to a game-state c in $Z \times Q$ combined with an element p from $\bar{\mathcal{L}}$.*

Suppose the refinement heuristic Heuristic yields (c, p) . Then c as well as the game states \tilde{c} forwardly/backwardly reachable from c via junction transitions are split by p

Algorithm PropertyCheck (A : pointed automata, T : rooted transition system)
Local variables P : an abstract property-game, initialized with $P_{T,A}^I$

- 1: Simplify (G)
- 2: while ($\omega(c^i) = \perp$) do
 - 3: Redirect every transition t pointing to a hyper-point $(\tilde{Z}, q) \in \mathbb{P}(Z) \times Q_{\text{qua}}$ such that it points to $(\pi_1(\{\tilde{Z}, q\}.R^-), q)$, where this hyper point (together with their maximal allowed outgoing transitions) is added to C (for example by using procedure Add).
% This step updates \tilde{Z} in case that some of the outgoing junction transitions of the hyper-point were removed during simplification. Note that the newly added game-states cannot be (in)valid.
 - 4: Remove from G every game-state $c \in C$ that is unreachable (from the initial game-state c^i), unless $c = (z, q) \in Z \times Q$ and there exists some reachable game-state $c' = (z', q')$ such that $\omega(c') = \perp$ and in addition, either $\delta(q') = \diamond q \wedge \omega(c) = \text{tt}$, or $\delta(q') = \square q \wedge \omega(c) = \text{ff}$. % Game-states that have no influence on (in)validity are removed. States fulfilling the last constraint are not removed, since they are needed for the computation of precise must hypertransitions in Refine.
- 5: Refine (P , Heuristic (P)) % Heuristic(P) yields a game-state combined with an element from \bar{C}
- 6: Simplify (G)
- 7: return $\omega(c^i)$

Table 4. A model checking algorithm for μ -calculus properties, where refinement is made locally on configurations, i.e., on abstract-states combined with properties. Here, the components of P and G are denoted as in Definition 6, resp. 1. Procedure Simplify is given in Table 3, Refine in Table 5, Add is explained in Section 4, and Heuristic is discussed in Section 5

in the abstract property-game. The transitions incoming/leaving such a new game-state \tilde{c}' split from \tilde{c} are calculated by taking the transitions incoming/leaving \tilde{c} into account. This procedure of simplification and local refinement is repeated until the property for the initial game-state is verified or falsified. The verification algorithm PropertyCheck is presented in Table 4 and its used Refine-procedure, which calculates the local refinement, is presented in Table 5. Note that the initial abstraction in PropertyCheck can be imprecise (if every concrete state has an outgoing transition or if none of them has one), but this imprecision will be eliminated after refinement steps.

In the following, the Refine-procedure is described in more detail. Its used procedures are informally described below, with the pseudo codes of OutgoingMayCalculation and OutgoingMustCalculation given in Appendix A. Let (c, p) , where $c = (z, q)$, be the game-state and predicate returned by Heuristic. In Line 1, the new abstract-states z_1 and z_2 are determined, as the result of splitting z based on p . Here the abstract states and ϱ are encoded as in cartesian predicate abstraction, i.e., an abstract-state is a function from a set of predicates into a three valued domain, indicating whether the corresponding predicate is used, its negation is used, or is not considered. Consequently, if suitable refinement heuristics (e.g., those presented in Section 5) are used, the resulting substates z_1 and z_2 can effectively be calculated.

Q' is used to collect the states that have to be split, i.e., are connected via junction transitions to c . It is sufficient to collect in Q' only automaton-states, since it is an invariant that the first component of game-states connected to c via junction transitions is always z . Set Q' is initialized to q , representing c . Every state \tilde{q} in Q' , representing

Algorithm Refine (P : an abstract property-game, $((z, q), p) : (Z \times Q) \times \bar{\mathcal{L}}$)
Local variables $Q' : \mathbb{P}(Q)$ initialized with $\{q\}$

- 1: Determine $z_1, z_2 \in Z$ (and possibly add those elements to Z and adapt ϱ) such that $\llbracket \varrho(z_1) \rrbracket = \llbracket \varrho(z) \wedge p \rrbracket$ and $\llbracket \varrho(z_2) \rrbracket = \llbracket \varrho(z) \wedge \neg p \rrbracket$
- 2: while $Q' \neq \{\}$ do
- 3: remove an element \tilde{q} from Q'
- 4: Add $(P, (z_1, \tilde{q}))$; Add $(P, (z_2, \tilde{q}))$ % Adding of the game-states obtained from splitting.
- 5: if $c^i = (z, \tilde{q})$ then (if **Satisfiable** ($p^i \wedge \varrho[z_1]$) then $c^i := (z_1, \tilde{q})$ else $c^i := (z_2, \tilde{q})$) % Relocation of the initial game-state.
- 6: if $\tilde{q} \notin Q_{\text{qua}}$ then
- 7: while $\{(z, \tilde{q}).R^- \neq \{\}$ do % Calculation of the outgoing junction transitions
- 8: remove an element (z', \tilde{q}') from $\{(z, \tilde{q}).R^-$ % By an invariant $z' = z$
- 9: $Q' := Q' \cup \{\tilde{q}'\} \setminus \{\tilde{q}\}$; $R^- := R^- \cup \{((z_1, \tilde{q}), (z_1, \tilde{q}')), ((z_2, \tilde{q}), (z_2, \tilde{q}'))\}$
- 10: else % $\tilde{q} \in Q_{\text{qua}}$
- 11: **OutgoingMayCalculation** ($P, z, z_1, z_2, \tilde{q}$)
- 12: **OutgoingMustCalculation** ($P, z, z_1, z_2, \tilde{q}$)
- 13: while $(R^-.\{(z, \tilde{q})\}) \cap (Z \times Q) \neq \{\}$ do % Calculation of the incoming junction transitions
- 14: remove an element (z', \tilde{q}') from $(R^-.\{(z, \tilde{q})\}) \cap (Z \times Q)$ % By an invariant $z' = z$
- 15: $Q' := Q' \cup \{\tilde{q}'\} \setminus \{\tilde{q}\}$; $R^- := R^- \cup \{((z_1, \tilde{q}'), (z_1, \tilde{q})), ((z_2, \tilde{q}'), (z_2, \tilde{q}))\}$
- 16: **IncomingMayCalculation** ($P, z, z_1, z_2, \tilde{q}$)
- 17: **IncomingMustCalculation** ($P, z, z_1, z_2, \tilde{q}$)
- 18: $C := C \setminus \{(z, \tilde{q})\}$; $C_1 := C_1 \setminus \{(z, \tilde{q})\}$; $C_2 := C_2 \setminus \{(z, \tilde{q})\}$

Table 5. An algorithm for local refinement calculation, where the components of P are denoted as in Definition 6, resp. 1. The occurring procedures are explained in Section 4.

the game-state $\tilde{c} = (z, \tilde{q})$, is split by splitting z to z_1 and z_2 . The resulting game-states are added to the abstract property-game, using **Add** (Line 4). If necessary, the initial game-state is recalculated via a satisfiability check that checks which of the substates of z abstracts s^i , characterized by p^i (Line 5).

After the substates of \tilde{c} are added as game-states, the transitions incoming/leaving \tilde{c} are recalculated, as ingoing/outgoing transitions of the new game states. Consider first the outgoing transitions (Lines 6-12). In case when $\tilde{q} \notin Q_{\text{qua}}$, the junction transitions leaving the game-state \tilde{c} being split are removed and correspondingly added to the two new game-states. Q' is extended with the target states (Line 9). In case when $\tilde{q} \in Q_{\text{qua}}$ the may as well as the must transitions leaving the new states are calculated by **OutgoingMayCalculation** and **OutgoingMustCalculation** resp. In this case the target game-states are not split (i.e., they are not added to Q'). This captures the laziness of the abstraction. In general, this step removes may-transitions that become redundant after refinement, as they do not represent any concrete transition. It also adds must transitions that did not exist before. It therefore makes the over and under approximations tighter.

More specifically, `OutgoingMayCalculation` checks if the may transition leaving $\tilde{c} = (z, \tilde{q})$ into $\tilde{c}' = (\tilde{z}', \text{succ}(\tilde{q}))$ also exists for the new states (z_i, \tilde{q}) . This is done by using a theorem prover to check if z_i and \tilde{z}' fulfill the may condition.

In `OutgoingMustCalculation`, hypertransitions leaving (z, \tilde{q}) are taken for the new states without calculation. This is because when the must condition holds for z and some \tilde{Z} , it is guaranteed to also hold for any substate of z , representing a subset of concrete states. In addition, a must transition from the new game-state (z_i, \tilde{q}) into the hyper point (\tilde{Z}, \tilde{q}) is ‘added’ if z_i and $\mathcal{U}_G(\tilde{q}) \cup \tilde{Z}$ fulfill the must condition, where $\mathcal{U}_G(\tilde{q})$ denotes the game-states that (depending on the type of \tilde{q}) are valid, resp. invalid, at the succeeding state of \tilde{q} (i.e., at $\text{succ}(\tilde{q})$). Formally, for $\tilde{q} \in Q_{\text{qua}}$,

$$\mathcal{U}_G(\tilde{q}) = \begin{cases} \{z \mid \omega(z, \tilde{q}') = \text{tt}\} & \text{if } \delta(\tilde{q}) = \diamond \tilde{q}' \\ \{z \mid \omega(z, \tilde{q}') = \text{ff}\} & \text{if } \delta(\tilde{q}) = \square \tilde{q}'. \end{cases}$$

The consideration of $\mathcal{U}_G(\tilde{q})$ when checking the must condition, although it is not part of the hyper-point, is sound and is made for maintaining precision. It can be viewed as a shortcut for first including $\mathcal{U}_G(\tilde{q})$ in the hyper-point, and then removing it during simplification. Checking the must condition involves checking implication. Implication $a \Rightarrow b$ is checked by checking unsatisfiability of $a \wedge \neg b$. In order to reduce the number of theorem prover calls, only those \tilde{Z} are considered that are subsets of the targets of the may transitions leaving the corresponding new game-state. Furthermore, \tilde{Z} is automatically not considered if a superset is already determined to not fulfill the must condition. Similarly, once \tilde{Z} is determined to be a hypertransition, none of its supersets is checked. This is justified by the fact that including only *minimal* sets \tilde{Z} as hyper-points does not damage precision [26].

Consider now the incoming transitions (Lines 13-17). The incoming junction transitions of \tilde{c} originating in game-states that are not hyper-points are calculated similarly to the outgoing junction transitions, where also Q' is extended (Line 15). The incoming may transitions are calculated, analogously to the outgoing may transitions, in `IncomingMayCalculation`, where may transitions can possibly be removed, making the overapproximation tighter.

The calculation of the incoming must transitions is made in `IncomingMustCalculation`. Here a difference arises compared to the outgoing must transitions. Since must transitions always lead to hyper-points, no must transition points directly to the split game-state $\tilde{c} = (z, \tilde{q})$, but a must transition can indirectly point to \tilde{c} via a hyper-point (\tilde{Z}', \tilde{q}') . We consider such must transitions as incoming must transitions. The hyper-point \tilde{Z}' that contains the abstract-state z being split is possibly refined (and made tighter) by keeping only one of the substates z_1 or z_2 in it. The existence of such a tighter hypertransition is checked (and resp. added) by checking if the must condition is fulfilled when replacing z by z_1 or z_2 . In case that none of these two refined hypertransitions exists, the one where z is replaced by both new states in \tilde{Z}' is added without a necessary calculation. Note that if a refined hypertransition is discovered, then the latter hypertransition is redundant (as it is less precise), and is hence not included. Compared to the calculation of the outgoing must transitions, where transitions could possibly be added, in this case we simply make the existing ingoing must transitions more precise.

Note that after the calculation of the outgoing may and must transitions, the game-state \tilde{c} being split (which will be removed in the end) is still allowed as target, i.e., it is possible that a new game-state can point to \tilde{c} . But after the recalculation of incoming may and must transitions, these cases, where \tilde{c} is the target are handled. Thus, when **Refine** terminates it is ensured that no transition incoming/leaving \tilde{c} can exist. In particular, self-loops are adequately refined by our approach.

New game-states are added with the **Add**-procedure, which is also responsible for updating the validity function ω . Procedure **Add** $(G, (\eta, \tilde{q}))$ adds to G the game-state $(\eta, \tilde{q}) \in (Z \times Q) \cup (\mathbb{P}(Z) \times Q_{\text{qua}})$, if it is not already present, such that it yields an abstract property-game. In particular, if $(\eta, \tilde{q}) \in \mathbb{P}(Z) \times Q_{\text{qua}}$, then all possible transitions leaving the new hyper-point to $\{(z, \text{succ}(\tilde{q})) \mid z \in \eta\}$ are also added. Furthermore, if the automaton component \tilde{q} of an added game-state $(\eta, \tilde{q}) \in Z \times Q$ is such that $\delta(\tilde{q})$ is a predicate in \mathcal{L} , then ω is determined at it by calculating if $\varrho(\eta) \Rightarrow \delta(\tilde{q})$ or $\varrho(\eta) \Rightarrow \neg\delta(\tilde{q})$ holds. Again, implication is checked via the equivalent unsatisfiability check.

Example 1. Consider the abstract property-game depicted in Figure 2 (e), where a refinement heuristic determined that the game-state $c = (\ell = 1 \wedge x \neq 1, \square)$ needs to split according to the predicate $(\ell = 0 \wedge x \neq 0) \vee (\ell = 1 \wedge x > 2)$. Figure 2 (f) depicts the result of the local refinement. Initially, c is split into $(\ell = 1 \wedge x \in \{0, 2\}, \square)$ and $(\ell = 1 \wedge x > 2, \square)$. The outgoing transitions of the substates are recalculated: c has only two outgoing may transitions, pointing to $(\ell = 1 \wedge x = 1, \wedge)$ and $(\ell = 1 \wedge x \neq 1, \wedge)$. The first remains as an outgoing may transitions of $(\ell = 1 \wedge x \in \{0, 2\}, \square)$, while the second remains as an outgoing may transition of $(\ell = 1 \wedge x > 2, \square)$. The latter transition is also added as an outgoing must transition of $(\ell = 1 \wedge x > 2, \square)$, as it now fulfills the must condition (more precisely, a hyper-point $(\{\ell = 1 \wedge x \neq 1\}, \square)$ is added, with an incoming must transition from $(\ell = 1 \wedge x > 2, \square)$, and outgoing junction transition to $(\ell = 1 \wedge x \neq 1, \wedge)$). Next, the incoming transitions of c are considered. As a result, the source state, $\tilde{c} = (\ell = 1 \wedge x \neq 1, \wedge)$, of the incoming junction transition of c is also split into $(\ell = 1 \wedge x \in \{0, 2\}, \wedge)$ and $(\ell = 1 \wedge x > 2, \wedge)$. The junction transitions are adapted accordingly, and the rest of the transitions of the substates of \tilde{c} are calculated: the incoming may transitions of \tilde{c} become incoming transitions of both its substates. In addition, the incoming must transition of \tilde{c} from $(\ell = 1 \wedge x > 2, \square)$, that was added during the refinement, becomes a must hypertransition (more precisely, the hyper-point which previously consisted of a singleton set $\{\ell = 1 \wedge x \neq 1\}$, now consists of the two abstract states to which $\ell = 1 \wedge x \neq 1$ was split, however, the hyper-point is omitted from the figure, and a must hypertransition is used instead).

So far some limitations exist in our model checking algorithm, restricting the practical relevance of the algorithm in its current version. Those points, as well as corresponding optimizations of the algorithm, are discussed in Section 6.

Theorem 2 (Soundness). *Suppose satisfiability checks are sound and complete and Heuristic is a refinement heuristic. If $\text{PropertyCheck}(A, T)$ returns tt (ff) then $T \models q$ (resp. $T \not\models q$) holds.*

Theorem 3 (Relative completeness). *Suppose satisfiability checks are sound and complete and \mathcal{L} can describe every subset of S . If the acceptance function of A always maps*

to zero (i.e., A corresponds to a least fixpoint free μ -calculus formula) and $T \models q$, then there exists a (not necessarily computable) refinement heuristic **Heuristic** such that $\text{PropertyCheck}(A, T)$ returns `tt`.

Note that the usage of hypertransitions is necessary for Theorem 3, since allowing only singleton targets yields a model that is not complete for safety-properties with respect to predicate abstractions, see, e.g., [6]. Theorem 3 does not hold if we restrict to computable refinement heuristics, since otherwise the halting problem would be decidable. Furthermore, Theorem 3 does not hold for automata with arbitrary acceptance function, since the underlying class of abstract models is not expressive enough. Fairness constraints, as in [4, 7], are needed.

5 Heuristics

The CEGAR-based algorithm described in Section 4 uses a refinement heuristic to determine a game-state c that should be split, and a predicate p , according to which c is split, along with the game-states reachable from it via junction transitions. In this section we define the special class of pre-based heuristics and thereafter present and discuss suitable ones.

Definition 9. *Suppose P is an abstract property-game. Then a state $(z, q) \in C$ is predicate-unknown if $\delta(q) \in \mathcal{L}$ and $\omega(z, q) = \perp$. A real may transition is a $t \in R^+$ that has no corresponding must transition, more precisely, every must transition $t' \in R^-$ that leaves the same source ($\text{sor}(t) = \text{sor}(t')$) has a target $\text{tar}(t')$ whose first component \tilde{Z} is different from the singleton set consisting of the first component z of the target of t (i.e., $\pi_1(\text{tar}(t')) \neq \{\pi_1(\text{tar}(t))\}$).*

A refinement heuristic **Heuristic** is pre-based if the return value is derived from a predicate-unknown state or from a real may transition, whenever one of them exists. More precisely, if $\text{Heuristic}(P) = (c, p)$ then (i) $c = (z, q)$ is a predicate-unknown state in C and $p = \delta(q)$ or (ii) $c = \text{sor}(t)$ for some real may transition $t \in R^+$ and $p = \text{pre}(\varrho(\pi_1(\text{tar}(t))))$ or (iii) neither a predicate-unknown state nor a real may transition exists.

Proposition 2. *A simplified abstract property-game, where the initial game-state is neither valid nor invalid, i.e. $\omega(c^i) = \perp$, has a predicate-unknown state or a real may transition.*

Intuitively, predicate-unknown states and real may transitions are good candidates for refinement since they can be viewed as a cause for uncertainty. Pre-based refinement heuristics are sufficient for finite state systems:

Theorem 4 (Termination). *Suppose T has a finite bisimulation quotient (with respect to the elements of \mathcal{L} that occur in A), satisfiability checks are sound and complete, and **Heuristic** is a pre-based refinement heuristic. Then $\text{PropertyCheck}(A, T)$ terminates, i.e., returns `tt` or `ff`.*

5.1 Bottom up strategy

Determine (i) a predicate-unknown state (z, q) or (ii) a real may transition t that points to an (in)valid game-state, i.e., $\omega(\text{tar}(t)) \neq \perp$. Return $((z, q), \delta(q))$, resp. $(\text{sor}(t), \text{pre}(\varrho(\pi_1(\text{tar}(t))))$). Note that such states, resp. real may transitions, do not always exist in simplified abstract property-games. In such a case an arbitrary real may transition t is chosen. An advantage of the bottom up strategy is that (if case (i) or (ii) are applicable) at least one of the new game-states is (in)valid after the refinement. A disadvantage of the bottom up strategy is that it can become an unnecessary source of nontermination:

Example 2. Consider the example from Section 2. Then the bottom up strategy will ‘run to’ Figure 2 (e) and then determine the may transition pointing to the invalid state. Since $\text{pre}(\varrho(\ell = 1 \wedge x = 1)) = (\ell = 1 \wedge x = 2) \vee (\ell = 0 \wedge x = 0)$, the result of refinement will be splitting the source state $\ell = 1 \wedge x \neq 1$ to $\ell = 1 \wedge x = 2$ and $\ell = 1 \wedge x \neq 1 \wedge x \neq 2$. After simplification, an abstract property-game equivalent to (e), which is already equivalent to (c), will be generated (with the abstract state $\ell = 1 \wedge x \neq 1$ replaced by $\ell = 1 \wedge x \neq 1 \wedge x \neq 2$, and the abstract state $\ell = 1 \wedge x = 1$ replaced by $\ell = 1 \wedge x = 2$). This will continue forever, replacing $\ell = 1 \wedge x \neq 1$ by $\ell = 1 \wedge x \neq 1 \wedge \dots \wedge x \neq i$, and $\ell = 1 \wedge x = 1$ by $\ell = 1 \wedge x = i$.

5.2 Breadth first strategy

Determine a state (z, q) that (i) is a predicate-unknown state or a source of a real may transition t and (ii) has a minimal distance to the initial game-state. Return $((z, q), \delta(q))$, resp. $(\text{sor}(t), \text{pre}(\varrho(\pi_1(\text{tar}(t))))$). Note that it is possible that after the refinement step, the distance of the next witness state (z, q) will decrease, since a must transition ‘pointing’ to $\{(z', q')\}$ can become a hypertransition, pointing to $\{(z'_1, q'), (z'_2, q')\}$, resulting in a real may transition pointing, e.g., to (z'_1, q') , whereas the original may transition pointing to (z', q') was not a real one.

Example 3. Consider the example from Section 2. Then the breadth first strategy will split one of the (true, \diamond) states in Figure 2 (a) along the weakest precondition of true , as it is made in (f). Thus the property will be shown after a single refinement step. The success of the breadth first strategy in this example is due to the shallow depth of the loop $q_0 \rightarrow q_5 \rightarrow q_0$, which ensures that this strategy manages to recognize the infinite must path and thus it finds the property to be valid. But if, e.g., the property of Figure 1 is transformed into the (equivalent) property where the loop $q_0 \rightarrow q_5 \rightarrow q_0$ is replaced by a ‘deeper’ loop $q_0 \rightarrow q_5 \rightarrow q_6 \rightarrow \dots \rightarrow q_n \rightarrow q_0$ in which q_6, \dots, q_n are also \diamond -states, then the depth of the loop makes the breadth first strategy run into the same live-lock described in Example 2 after the first few refinement steps, before it finds the infinite must path. Thus, it fails to terminate.

5.3 Youngest first strategy

Determine a state (z, q) that (i) is a predicate-unknown state or a target of real may transition t and (ii) is minimal with respect to the number of splits used to obtain z . Return $((z, q), \delta(q))$, resp. $(\text{sor}(t), \text{pre}(\varrho(\pi_1(\text{tar}(t))))$). Point (ii) can easily be determined if

the abstract states are encoded via the afore mentioned cartesian predicate approach, since only the positions where the cartesian function does not map to ‘unused’ have to be counted. Note that this kind of heuristics cannot be defined, if a global refinement approach is used, where every state is split by the new predicate.

Example 4. In Figure 2 (a) the youngest first strategy will split either the source state of one of the three real may transitions along the weakest precondition of $true$ or the state $(true, \diamond)$. If the split state in the first refinement step is one of the two $(true, \diamond)$ states, then in particular the initial state is split and the property will be shown. If one of the other two states is split, then both of them are split, and so is the target of the real may transition leaving the upper $(true, \diamond)$ state (as these states are connected via junction transitions). This ensures that the lower $(true, \diamond)$ state, whose outgoing real may transition leads to a yet unsplit state, will be split along the weakest precondition of $true$ in the second refinement step. Thus, at latest in the second refinement step the initial state will be split along the weakest precondition of $true$, and the property will be shown. The youngest first strategy also succeeds for the modified property described in Example 3.

In order to maintain the advantage of the bottom up strategy, real may transitions to (in)valid states can be restrictively favored by, e.g., doubling the ‘age’ of the states that are unknown. Sometimes pre-based refinement heuristics are not sufficient:

Example 5. Consider the property $\rightarrow \boxed{\diamond}$ checked on system $\xrightarrow{x:=0} \text{[circle]} \begin{matrix} [x=0] \\ [x>1] \end{matrix} \xrightarrow{x:=x-1}$,

where $x \in \mathbb{N}$. Then any pre-based refinement heuristics will produce after n refinements the (simplified) abstract property-game $\rightarrow \boxed{x=0 \vee x > n} \diamond$, i.e., the property cannot

be verified. On the other hand if first the initial state is separated, then $\rightarrow \boxed{x=0} \diamond \quad \boxed{x > 1} \diamond$ is obtained. Thus the property can be shown.

6 Optimizations of the algorithm

For the sake of completeness, we present some possible optimizations of the algorithm.

Too many Simplify calls. The Simplify procedure is called after every local refinement. Thus an expensive algorithm is calculated, while expecting only small improvements, since only a local refinement was made. To remedy this, more refinement iterations can be made before Simplify is called again. Further optimization is obtained, if the validity function is also adapted during the refinement calculation, e.g., by backwards search when a state is determined to become (in)valid, and Simplify is only called for more exact determination of least fixpoint properties.

No reuse of theorem prover calls. Typically the same satisfiability checks are calculated multiple times, since they (mainly) depend on the abstract state and not on the

property of the configuration. Therefore, those calls can be reused by caching, or by using an additional generalized Kripke modal transition system, where the abstract states and their may and must (hyper)transitions are stored, resp. negatively stored, whenever a corresponding satisfiability check is made. Here, a tradeoff between time and space arises. Furthermore, the heuristics can be tuned to prefer those game-states for which no (or less) new satisfiability checks have to be made to determine the refinement.

Unnecessary many theorem prover calls. In case of refinement, a forward search of game-states reachable via junction transitions always takes place. However, it is possible that the current refined game-state will immediately become (in)valid (e.g., due to the first optimization) and thus the (in)validity of its reachable game-states will be irrelevant. Therefore, such a forward search should only take place if the validity of the current game-state cannot be determined immediately.

Complex $\text{pre}(\psi)$ -calculations. The algorithm starts with the most general abstraction consisting of only one abstract state, thus coarse abstractions arise. Such abstractions have the disadvantage that the calculation of $\text{pre}(\psi)$ is in general expensive. Therefore, it is beneficial to start with a less coarse initial abstraction, which can be determined by pre-examination of the underlying systems (e.g., by partitioning the code-lines). The techniques of interpolations [13, 22] might also help to avoid the high cost of $\text{pre}(\psi)$ -calculations.

Too complex formulas for the theorem prover. Due to satisfiability checks of complex formulas, the calculation time of the theorem prover can outweigh the calculation time of the parity game algorithm. A remedy is to drop precision and use further approximations. A refinement step can, in addition to the extension of the abstract state space, perform a more precise calculation of the used approximations. We suggest the following approximations:

- Approximate the predicates of the different states by using two formulas: one for an over and the other for an under approximation of the precise formula. In each calculation those approximation formulas that guarantee soundness are used.
- Approximate the must transitions, i.e., only calculate a subset of possible must transitions. For example, first calculate those having a single target and as a refinement step calculate those having two elements as target, etc. Alternatively, only calculate the hypertransitions on demand inside the parity game algorithm, as in [25].
- Approximate the system, e.g., instead of using T , use an approximated system for which $\text{pre}(\psi)$ can be more efficiently calculated.
- Approximate the theorem prover queries by clustering predicates [16]. In this approach, one theorem prover call is split into many having less complex formulas and their results are combined afterwards, where precision is lost.

How exactly these approximation techniques can be applied is a topic of future work.

7 Conclusion

We presented a new CEGAR-based algorithm for the μ -calculus, where refinement is local and the refinement determination is separated from the model checking algorithm. Three different refinement heuristics are developed, where the most promising

one heavily depends on the local refinement approach. It is even possible that our algorithm will yield improvements for safety properties, since by using a 3-valued abstract model better refinement heuristics can be obtained. Exact examinations will take place after the implementation of our algorithm, which is future work. The investigation of other refinement heuristics is also the subject of future work.

References

1. T. Ball and O. Kupferman. An abstraction-refinement framework for multi-agent systems. In *LICS*. IEEE Computer Society Press, 2006.
2. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
4. D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *LICS*, pages 335–344. IEEE Computer Society Press, 2004.
5. L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *LICS*, pages 170–179, 2004.
6. H. Fecher and M. Huth. Complete abstraction through extensions of disjunctive modal transition systems. Technical Report 0604, Christian-Albrechts-Universität zu Kiel, 2006.
7. H. Fecher and M. Huth. Ranked predicate abstraction for branching time: Complete, incremental, and precise. In *ATVA*, volume 4218 of *LNCS*, pages 322–336. Springer, 2006.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
9. O. Grumberg, M. Lange, M. Leucker, and S. Shoham. *Don't know* in the μ -calculus. In *VMCAI*, volume 3385 of *LNCS*, pages 233–249. Springer, 2005.
10. O. Grumberg, M. Lange, M. Leucker, and S. Shoham. When not losing is better than winning: Abstraction and refinement for the full μ -calculus. *Information and Computation*, 2007. doi: 10.1016/j.ic.2006.10.009.
11. B. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE*. ACM, 2006.
12. A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In *TACAS*, 2006.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
15. M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *ESOP*, volume 2028 of *LNCS*, pages 155–169, 2001.
16. H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *DAC*, pages 445–450. ACM, 2005.
17. M. Jurdzinski. Deciding the winner in parity games is in $UP \cap co-UP$. *Inf. Process. Lett.*, 68(3):119–124, 1998.
18. H. Klauck. Algorithms for parity games. In *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*, pages 107–129. Springer, 2002.
19. D. Kozen. Results on the propositional μ -calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
20. K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.
21. K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society Press, 1990.
22. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
23. K. Namjoshi. Abstraction for branching time properties. In *CAV*, pages 288–300, 2003.

24. A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *DAC*, pages 457–462, 1998.
25. S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. In *LICS'06*.
26. S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS*, 2004.
27. Th. Wilke. Alternating tree automata, parity games, and modal μ -calculus. *Bull. Soc. Math. Belg.*, 8(2):359–391, May 2001.
28. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.

A Pseudo code of additional procedures

Algorithm `OutgoingMayCalculation` ($P, z, z_1, z_2, \tilde{q}$) % calculation of the weak transitions leaving (z_1, \tilde{q}) or (z_2, \tilde{q})

```

1: while  $\{(z, \tilde{q})\}.R^+ \neq \{\}$  do
2:   remove an element  $(\tilde{z}', \tilde{q}')$  from  $\{(z, \tilde{q})\}.R^+$ 
3:   if Satisfiable  $(\varrho(z_1) \wedge \text{pre}(\varrho(\tilde{z}')))$  then  $R^+ := R^+ \cup \{(z_1, \tilde{q}), (\tilde{z}', \tilde{q}')\}$ 
4:   if Satisfiable  $(\varrho(z_2) \wedge \text{pre}(\varrho(\tilde{z}')))$  then  $R^+ := R^+ \cup \{(z_2, \tilde{q}), (\tilde{z}', \tilde{q}')\}$ 
5: od

```

Algorithm `OutgoingMustCalculation` ($P, z, z_1, z_2, \tilde{q}$) % calculation of the must transitions leaving (z_1, \tilde{q}) or (z_2, \tilde{q})

Local variables $M_1, M_2, N_1, N_2 : \mathbb{P}(Z)$ % M_i stores the yet determined relevant must transition for (z_i, \tilde{q}) ; N_i stores the not yet considered, relevant must transition for (z_i, \tilde{q})

```

1:  $M_1 = \{\tilde{Z} \cap \pi_1(\{(z_1, \tilde{q})\}.R^+) \mid \tilde{Z} \in \pi_1(\{(z, \tilde{q})\}.R^-)\}$ 
2:  $M_2 = \{\tilde{Z} \cap \pi_1(\{(z_2, \tilde{q})\}.R^+) \mid \tilde{Z} \in \pi_1(\{(z, \tilde{q})\}.R^-)\}$ 
3: remove all elements from  $\{(z, \tilde{q})\}.R^-$ 
4:  $N_1 := \pi_1(\mathbb{P}(\{(z_1, \tilde{q})\}.R^+)) \setminus \{\tilde{Z}_1 \mid \exists \tilde{Z} \in M_1 : \tilde{Z} \subseteq \tilde{Z}_1\}$ 
5:  $N_2 := \pi_1(\mathbb{P}(\{(z_2, \tilde{q})\}.R^+)) \setminus \{\tilde{Z}_2 \mid \exists \tilde{Z} \in M_2 : \tilde{Z} \subseteq \tilde{Z}_2\}$ 
6: for i=1 to 2 do
7:   while  $N_i \neq \{\}$  do
8:     take (not remove) an element  $\tilde{Z}$  from  $N_i$ 
     % Check if a must transition exists (iff  $\varrho(z_i) \Rightarrow \text{pre}(\bigvee_{\tilde{z} \in \mathcal{U}_G(\tilde{q}) \cup \tilde{Z}} \varrho(\tilde{z}))$ ) by using a satisfiability check
9:     if  $\neg$  (Satisfiable  $(\varrho(z_i) \wedge \neg(\text{pre}(\bigvee_{\tilde{z} \in \mathcal{U}_G(\tilde{q}) \cup \tilde{Z}} \varrho(\tilde{z}))))$ ) then
10:       $M_i := M_i \cup \{\tilde{Z}\} \setminus \{\tilde{Z}_i \mid \tilde{Z} \subset \tilde{Z}_i\}$  ;  $N_i := N_i \setminus \{\tilde{Z}_i \mid \tilde{Z} \subseteq \tilde{Z}_i\}$ 
11:     else  $N_i := N_i \setminus \{\tilde{Z}_i \mid \tilde{Z}_i \subseteq \tilde{Z}\}$ 
12:   od
13: For  $\tilde{Z} \in M_i$  do % Add the calculated hyper-transitions
14:   Add  $(P, (\tilde{Z}, \tilde{q}))$  ;  $R^- := R^- \cup \{(z_i, \tilde{q}), (\tilde{Z}, \tilde{q})\}$ 
15: next
16: next

```

Table 6. Procedures for the calculation of the outgoing may, resp. must, transitions