Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Local abstraction-refinement for the mu-calculus*

Harald Fecher[1], **Sharon Shoham**[2]

[1] Albert-Ludwigs-Universität Freiburg, Germany
[2] The Technion, Haifa, Israel

Received: date / Revised version: date

Abstract. A key technique for the verification of programs is counterexample-guided abstraction refinement (CEGAR). 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

Model checking [5,35] provides a framework for verifying properties of systems: a system is represented by a mathematical model M, a property of interest is coded within a formal language as some ϕ, and satisfaction is a formally defined predicate $M \models \phi$. The fact that the size of M is often exponential or even infinite in its description, which could be a computer program, often requires alternatives to the direct computation of $M \models \phi$.

One of the most successful techniques for checking correctness of large or even infinite programs is predicate abstraction [15] with *counterexample-guided abstraction refinement* (CEGAR) [6]. This approach consists of three phases: abstraction, model checking, and refinement. A typical tool based on that technique is SLAM [3], 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 [21], 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. Lazy abstraction has the advantage that the abstract system does not grow where it is not needed. Both tools mentioned are only capable of verifying safety properties.

Grumberg et al. [16,17] present CEGAR-based algorithms for the verification of the μ-calculus [28], which is a powerful formalism for expressing branching time[1] and reachability properties by using fixpoint constructions. These approaches use as underlying abstract models *Kripke modal transition systems* [22], which have may and must transitions (over, resp., under approximation of the concrete transitions), as in *modal transition systems* [30]. Two transition relations are essential in order to preserve branching time properties. They also enable to preserve both *validity* and *invalidity* from the abstract model to the concrete model, at the cost of introducing a third truth value *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 particular, the 3-valuedness makes it unnecessary to check spuriousness of a counterexample, since when a counterexample is found, it is always a real one.

In [16], a 3-valued satisfaction game for the μ-calculus is defined, where the Verifier tries to obtain validity, and

* A preliminary version of this work appeared in [12].

[1] Branching time is relevant whenever nondeterminism occurs from external factors (e.g., user input), from random behavior, from the modeling of faulty systems or channels, or from the abstraction of time or communication arguments [11].

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 [39], determines an abstract state z and a split criterion such that the splitting of z with respect to the split criterion leads to less spurious behavior. This approach is generalized in [17] by making the approach independent from the Zielonka algorithm, i.e., allowing more efficient algorithms [27]. 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 abstract state to split, as well as the split criterion, are 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 [16,17] in the following way:

- A more expressive underlying abstract model is used, namely *generalized Kripke modal transition systems* [36], where must hypertransitions, as in *disjunctive modal transition systems* [31], are used, i.e., a must transition points to a set of states rather than to a singleton[2]. Consequently, a smoother refinement determination can be obtained [36] and more properties can in principle be shown [9].
- 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.
- Our algorithm is integrated with *predicate abstraction*, making it more explicit, unlike the algorithm of [16,17], which uses a general notion of abstraction, and leaves the details on how a split is performed to the choice of the abstraction.

[2] In fact, we do not define the abstract system separately. Instead, its description is intertwined with the property. Still, the underlying abstract system has the components of a generalized Kripke modal transition system.

Further related work. A CEGAR-approach to branching time properties is given in [34], where, contrary to our approach, only the transition relation is under, resp., over approximated (the state space remains equal). In [19], the techniques used in SLAM are generalized to branching time properties, where the underlying abstract model is equivalent to Kripke modal transition systems.

In [33] 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 [8]. 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 model checking algorithm, i.e., no separation is used. Must and may hypertransitions are also used in [10], 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 [37] 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. [37] also suggests a CEGAR-based algorithm, however they consider only the alternation-free fragment of the μ-calculus. Moreover, their refinement follows [16], resulting in a weak form of lazy abstraction.

Another possibility to increase expressiveness without using hypertransitions is given in [2]: Backward must-transitions and entry/exit points are used to conclude the existence of transitive must-transitions.

In [18] 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 newly developed 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.

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.

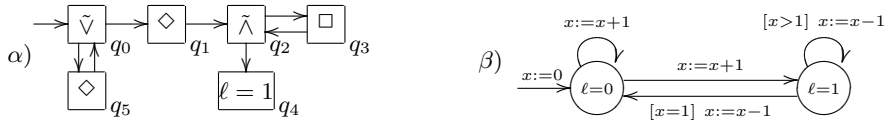


Fig. 1. A μ-calculus formula (α) in terms of automata (see Section 3.3), and a system (β). α): The property at the initial state q_0 holds if (i) there is a transition such that $\ell = 1$ holds on every possible path or (ii) there is a transition such that q_0 holds again (consequently, if there is 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. In the initial system state ($\ell = 0$), variable x is increased by 1 and it is nondeterministically decided whether to continue this behavior (remain in the initial system state) or to decrease x by 1 step by step until the value of x becomes 1, in which case the initial system state is reached again.

The model checking is based on a configuration structure (later referred to as an abstract property-game), where each configuration (later referred to as a game-state), consists of a subproperty and a (possibly abstract) state of the system. The transitions reflect the dependencies between configurations: 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 $\tilde{\wedge}, \tilde{\vee}, \Diamond, \Box$. Intuitively, $\tilde{\wedge}$ and $\tilde{\vee}$ stand for the logical connectives $\wedge$ and $\vee$ resp. Similarly, $\Diamond$ stands for "exists a successor", while $\Box$ stands for "all successors".

The first configuration structure is obtained by combining all subproperties (automaton states) with the single abstract state *true*, abstracting any concrete system-state (in practice, a finer initial abstraction will be used). In addition, the transition relation of the system is over-approximated 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. The (in)validity of all the configurations is initially unknown. May and must transitions, which represent the system's transitions, leave $\Diamond$- or $\Box$-configurations. In such configurations the transitions of the system need to be considered in order to determine the value of the subproperty. The other transitions (that leave $\tilde{\wedge}$- or $\tilde{\vee}$-configurations) are called *junction transitions*. Junction transitions imitate the automaton transitions.

In general, the algorithm iterates four phases: validity and invalidity 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 $\tilde{\vee}$- and $\Diamond$-configurations, and the Falsifier moves in $\tilde{\wedge}$- and $\Box$-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 $(\mathit{true}, \ell = 1)$, 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 $(\mathit{true}, \Diamond)$ to $(\mathit{true}, \vee)$ 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 predicate configuration $(\mathit{true}, \ell = 1)$, whose validity is unknown, then all configurations forwardly/backwardly reachable from $(\mathit{true}, \ell = 1)$ 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 [36], 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 $\widetilde{Z}$ exists iff every concrete state abstracted by z_1 has a transition with a target that is abstracted by an element from $\widetilde{Z}$. Consequently, *true* has an outgoing may transition, but no must transition (no transition is enabled in the system at $\ell = 1 \wedge x = 0$). Thereby, the configuration structure from Figure 2 (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 $(\mathit{true}, \ell = 1)$ 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-

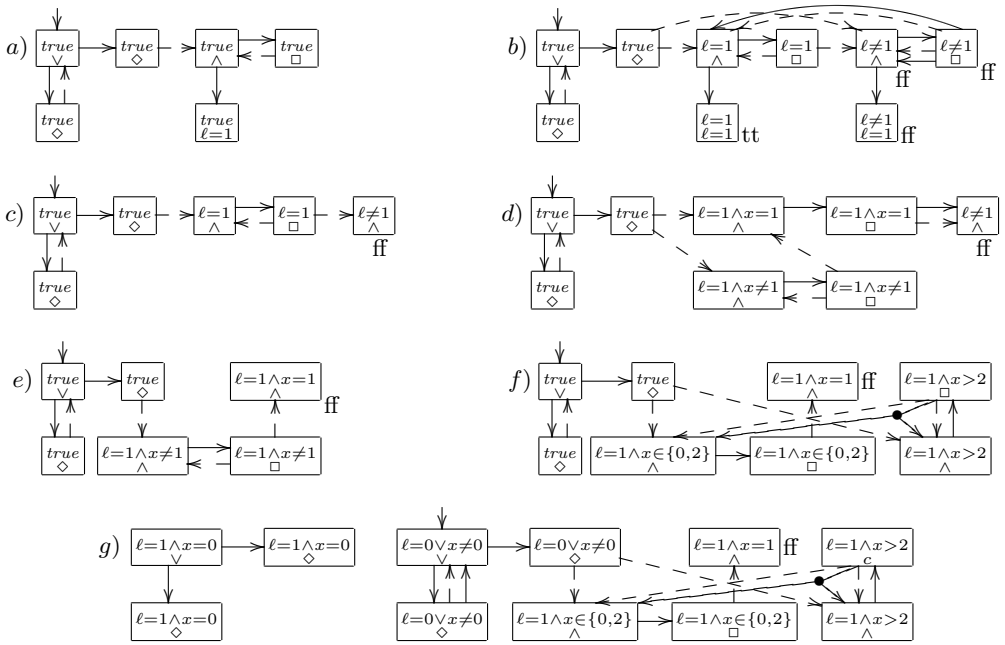


Fig. 2. Example of a property check via local refinement. May transitions are depicted as dashed arrows and must, as well as junction, transitions as solid arrows.

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 the configuration structure from Figure 2 (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 $\wedge$ dependent 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 $\ell = 0 \vee (x = 1 \wedge \ell = 1)$. The system state which appears in the source configuration is given by the predicate $\ell = 1$. It is therefore split to $\ell = 1 \wedge (\ell = 0 \vee (x = 1 \wedge \ell = 1))$ and to $\ell = 1 \wedge \neg(\ell = 0 \vee (x = 1 \wedge \ell = 1))$. Thus we obtain the configuration structure from Figure 2 (d), where we use the fact that $\ell = 1 \wedge (\ell = 0 \vee (x = 1 \wedge \ell = 1))$ is equivalent to $\ell = 1 \wedge x = 1$ and $\ell = 1 \wedge \neg(\ell = 0 \vee (x = 1 \wedge \ell = 1))$ is equivalent to $\ell = 1 \wedge x \neq 1$. Proceeding with (in)validity-determinations and simplifications, we obtain the configuration structure from Figure 2 (e).

Assuming the heuristic yields the may transition from $(\ell = 1 \wedge x \neq 1, \Box)$ 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 the configuration structure from Figure 2 (f), where a must hypertransition arises. Note that $\ell = 1 \wedge x \neq 1 \wedge ((\ell = 0 \wedge x \neq 0) \vee (\ell = 1 \wedge x > 2))$ is equivalent to $\ell = 1 \wedge x > 2$ and $\ell = 1 \wedge x \neq 1 \wedge \neg((\ell = 0 \wedge x \neq 0) \vee (\ell = 1 \wedge x > 2))$ is equivalent to $\ell = 1 \wedge x \in \{0, 2\}$. No further validity or invalidity can be determined in (f), thus no simplification takes place.

Assuming the heuristic yields the may transition into $(\mathit{true}, \vee)$, 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 the configuration structure from Figure 2 (g), where the initial configuration is also recalculated (since it was split). 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 $\bowtie \subseteq B \times D$ with subsets $X \subseteq B$ and $Y \subseteq D$ we write $b \bowtie d$ for $(b, d) \in \bowtie$ and $X.\bowtie$ for $\{d \in D \mid \exists b \in X\colon (b, d) \in \bowtie\}$ and $\bowtie.Y$ for $\{b \in B \mid \exists d \in Y\colon (b, d) \in \bowtie\}$. The projection to the i-th coordinate is denoted by π_i. Let $\mathrm{map}(f, \Phi)$ 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 sys-*

tem $T = (S, s^{\mathrm{i}}, \rightarrow, \mathcal{L})$ consists of a (possibly infinite) set S of states, an initial state $s^{\mathrm{i}} \in S$, 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 $[\![p]\!] \subseteq S$ of states), such that the following three conditions are satisfied. (i) There exists $p^{\mathrm{i}} \in \mathcal{L}$ with $[\![p^{\mathrm{i}}]\!] = \{s^{\mathrm{i}}\}$. (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 $\mathrm{pre}(\psi)$ of predicates from $\mathcal{L}$ such that $[\![\mathrm{pre}(\psi)]\!] = \rightarrow .[\![\psi]\!]$. In the following we assume a fixed rooted transition system $T = (S, s^{\mathrm{i}}, \rightarrow, \mathcal{L})$.

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 [17] by adding a validity function.

Definition 1. A *strong-weak-parity-game* $G = (C, C_1, C_2, c^{\mathrm{i}}, R^-, R^+, \theta, \omega)$ consists of
- 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 $c^{\mathrm{i}} \in C$,
- 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 $\omega : C \rightarrow \{\mathrm{tt}, \mathrm{ff}, \bot\}$, into the values true, false, and unknown.

The components of a strong-weak-parity-game G are denoted as in Definition 1. The source (target) of a transition t in G is denoted by $\mathrm{sor}(t)$, resp. $\mathrm{tar}(t)$. The validity function ω encodes an initial knowledge of the values of the game-states. Next we define validity and invalidity of a strong-weak-parity-game.

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 $\sup(\mathrm{map}(\theta, \Phi))$ 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 $\sup(\mathrm{map}(\theta, \Phi))$ 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}.

$\omega(c) \neq \bot$ or $c \notin C_1 \cup C_2$: Player 1 wins iff $\omega(c) = \mathrm{tt}$
$c \in C_1$ and $\omega(c) = \bot$: Player 1 picks as next game-state $c' \in \{c\}.R^-$;
$c \in C_2$ and $\omega(c) = \bot$: Player 2 picks as next game-state $c' \in \{c\}.(R^- \cup R^+)$;

Table 1. Moves of validity play 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.

$\omega(c) \neq \bot$ or $c \notin C_1 \cup C_2$: Player 2 wins iff $\omega(c) = \mathrm{ff}$
$c \in C_1$ and $\omega(c) = \bot$: Player 1 picks as next game-state $c' \in \{c\}.(R^- \cup R^+)$;
$c \in C_2$ and $\omega(c) = \bot$: Player 2 picks as next game-state $c' \in \{c\}.R^-$;

Table 2. Moves of invalidity play 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.

Remark 1. The validity, as well as the invalidity, game obviously corresponds to a parity game. Therefore, decidability of validity, resp. invalidity, is in UP ∩ coUP [26]. For an overview of algorithms for parity games we refer the reader to [29, 27].

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

Proof. It is easily seen that every winning strategy for Player 1 (Player 2) in the validity (resp. invalidity) game, which establishes that the strong-weak-parity-game is valid (resp. invalid), is also a winning strategy in the invalidity (resp. validity) game, refuting invalidity (resp. validity) of the strong-weak-parity-game. Therefore, a game cannot be both valid and invalid, which establishes the statement. □

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

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'.*

Proof. We first show that after every step of Simplify an equivalent strong-weak-parity-game is obtained. The fact that a strong-weak-parity-game is obtained is clear. The fact that after the validity (resp. invalidity) determination an equivalent game is obtained follows from the fact that a winning strategy before the (in)validity determination is also a winning strategy afterwards and vice versa. The removal of the transitions also has no influence on winning strategies, since the removed transitions will never be used in a winning strategy of the corresponding player, thus an equivalent strong-weak-parity-game is obtained. It is easy to verify that it is also a simplified one due to the adaptation of ω and since all the transitions that interfere with requirements (a)–(c) of Definition 3 are removed. Termination of Simplify follows from the termination of the parity game algorithms. □

3.3 Property language

We will present the modal μ-calculus [28] in its equivalent form of alternating tree automata [38] and define satisfaction via games, instead of using an inductive definition. This presentational choice simplifies proofs and anticipates future extensions of this work to fairness constraints in refinements as seen, e.g., in [7,10].

Definition 4 (Tree automata). An *alternating tree automaton* $A = (Q, q^{\mathrm{i}}, \delta, \Theta)$ has
- a finite, nonempty set of states $(q \in)Q$ with the initial element $q^{\mathrm{i}} \in Q$
- a transition relation δ mapping automaton states to one of the following forms, where q, q_1, q_2 are automaton states and $p \in \mathcal{L}$: $p \mid q \mid q_1 \tilde{\wedge} q_2 \mid q_1 \tilde{\vee} q_2 \mid \Diamond q \mid \Box q$
- an acceptance condition $\Theta\colon Q \to \mathbb{N}$ with finite image.

Such an automaton encodes a μ-calculus formula. Each automaton state q represents a subformula in accordance with $\delta(q)$. The operators $\tilde{\wedge}$ and $\tilde{\vee}$ represent the logical connectives $\wedge$ and $\vee$ resp. Similarly, $\Diamond$ and $\Box$ stand for the modalities "exists a successor", and "all successors" resp. The acceptance condition is used to express least and greatest fixpoints. 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 $A = (Q, q^{\mathrm{i}}, \delta, \Theta)$. Set Q_{qua} consists of those automaton-states of the form $\Diamond$ or $\Box$, i.e.,

$$Q_{\mathrm{qua}} = \{q \in Q \mid \exists q' : \delta(q) \in \{\Diamond q', \Box q'\}\}\,.$$

The successor state of $q \in Q_{\mathrm{qua}}$ is denoted by $\mathrm{succ}(q)$, i.e., $\mathrm{succ}(q) = q'$ if $\delta(q) \in \{\Diamond q', \Box q'\}$. 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

$$Q_2 = \{q \in Q \mid \delta(q) \in \bigcup_{q_1,q_2 \in Q} \{q_1 \tilde{\wedge} q_2, \Box q_1\}\}$$

those under control of Player 2. Satisfaction of a rooted transition system with respect to an alternating tree automaton 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 $(S \times Q, S \times Q_1, S \times Q_2, (s^{\mathrm{i}}, q^{\mathrm{i}}), R^-, \{\}, \Theta \circ \pi_2, \omega)$, where

$$R^- = \{((s,q),(s',q')) \mid (\delta(q) \in \{\Diamond q', \Box q'\} \wedge s \to s') \vee (s = s' \wedge \exists q'' : \delta(q) \in \{q', q' \tilde{\wedge} q'', q'' \tilde{\wedge} q', q' \tilde{\vee} q'', q'' \tilde{\vee} q'\})\}$$

$$\omega(s,q) = \begin{cases} \mathrm{tt} & \text{if } \delta(q) \in \mathcal{L} \wedge s \in [\![\delta(q)]\!] \\ \mathrm{ff} & \text{if } \delta(q) \in \mathcal{L} \wedge s \notin [\![\delta(q)]\!] \\ \bot & \text{otherwise.} \end{cases}$$

Furthermore, we write $T \models A$, whenever $P_{T,A}$ is valid, and otherwise, we write $T \not\models A$ (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 for the (in)validity evaluation of the game. Note that our definition of $T \models A$ coincides with the standard definition of satisfaction, and $T \not\models A$ coincides with the satisfaction of the dual formula, i.e., corresponds to negation.

Next, special strong-weak-parity-games that are derived for the satisfaction of alternating tree automata on abstracted systems, in the form of generalized Kripke modal transition systems [36], 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 $\widetilde{Z} \in \mathbb{P}(Z)$ are combined with Q_{qua}. The hyper-points are used to model hypertransitions. In this case, $\widetilde{Z}$ represents the disjunction of the abstract states $z' \in \widetilde{Z}$.

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 $G = (C, C_1, C_2, c^{\mathrm{i}}, R^-, R^+, \theta, \omega)$.

The classification of game-states to players is based on the property-states as before, except that in hyper-points the responsibilities of the players switch. For example, in a game-state of the form $(\widetilde{Z}, \Diamond q) \in \mathbb{P}(Z) \times Q_{\mathrm{qua}}$, Player 2 moves. Intuitively, this is because such a game-state is reached after Player 1 chose a hyper-transition from z to $\widetilde{Z}$, which led from the game-state $(z, \Diamond q) \in Z \times Q_{\mathrm{qua}}$ to $(\widetilde{Z}, \Diamond q)$. Since hypertransitions are interpreted disjunctively, it is now the role of the opponent, Player 2, to challenge the move of Player 1 by choosing an abstract state within $\widetilde{Z}$ to continue with.

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 for T and A is a tuple (Z, ϱ, G), where Z is a set of abstract states, $\varrho : Z \to \overline{\mathcal{L}}$ is an abstraction function, and G is a strong-weak-parity-game such that

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

P is *simplified* if G is.

The components of an abstract property-game P are denoted as in Definition 6. To simplify the presentation of the paper, we refrain from formalizing the additional properties of an abstract property-game. Instead, we describe them informally. The formal invariant can be found in the proof of Theorem 2. 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 systems, 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. $\varrho(z) \wedge \mathrm{pre}(\varrho(z'))$ is satisfiable, then there exists a may transition from (z, q) to $(z', \mathrm{succ}(q))$. This is called the *may condition*. A must transition from (z, q) to the hyper-point $(\widetilde{Z}, q)$ 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 $\widetilde{Z}$, i.e. the implication $\varrho(z) \Rightarrow \mathrm{pre}(\bigvee_{z' \in \widetilde{Z}} \varrho(z'))$ holds, or equivalently $\varrho(z) \wedge \neg \mathrm{pre}(\bigvee_{\tilde{z} \in \widetilde{Z}} \varrho(\tilde{z}))$ is unsatisfiable. The hyper-point $(\widetilde{Z}, q)$ is connected via junction transitions to the game-states in $\{(z', \mathrm{succ}(q)) \mid z' \in \widetilde{Z}\}$. However, simplification can damage these rules by removing some of the transitions.

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 $\widetilde{Z}$ 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, $\omega(z, q) = \mathrm{tt}$, resp. ff, if $\varrho(z) \Rightarrow p$, resp. $\varrho(z) \Rightarrow \neg p$, holds. Otherwise, $\omega(z, q) = \bot$. 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., $\varrho(z) = \mathit{true}$), corresponds to the abstract property-game given in the following definition.

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

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

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 $\bot$ 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 $(\widetilde{Z}, q)$ and junction transitions from the hyper-point to $\{(z', \mathrm{succ}(q)) \mid z' \in \widetilde{Z}\}$, Figure 2 directly connects (z, q) to $\{(z', \mathrm{succ}(q)) \mid z' \in \widetilde{Z}\}$ using a must transition, or, if necessary, a hypertransition.

4 CEGAR locally applied on configurations

Starting from the initial abstract property-game, a simplified abstract property-game is calculated by the verification algorithm. If the validity of the initial game-state remains unknown, i.e., $\omega(c^{\mathrm{i}}) = \bot$, then 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 $\overline{\mathcal{L}}$.

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 in the abstract property-game. In particular, these game-states all share the same abstract state, thus only one (and not every) abstract state is split via p. Moreover, only a subset of the game-states having this abstract state are refined. The transitions incoming/leaving 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 the Refine-procedure it uses, which calculates the local refinement, is presented in Table 5. We now elaborate further on these procedures.

The PropertyCheck-procedure starts by initializing the underlying abstract property-game P to the one defined in Definition 7. After a first Simplify call, a while loop is entered. The algorithm iterates the following phases as long as $\omega(c^{\mathrm{i}}) = \bot$, i.e., as long as the property is neither proved nor disproved: (i) some technical modifications of P are performed, (ii) the unreachable game-states that are also not needed later in the Refine-algorithm are removed from P, (iii) a refinement of P is calculated via the procedure Refine, and (iv) simplifications of P are made via Simplify. Further explanations are written as comments inside the algorithm. Note that the initial abstraction in PropertyCheck can be imprecise (if every concrete state has an outgoing transition, in which case a must transition should be added, or if none of them has one, in which case no may transition should be included), but this imprecision will be eliminated after refinement steps.

In the following, the Refine-procedure is described in more detail. The pseudo codes of its used procedures are given in Appendix A and are informally described below. 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 [14], 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 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 6-14). In case that $\tilde{q} \notin Q_{\mathrm{qua}}$, 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

Algorithm PropertyCheck (A : alternating tree automaton, T : rooted transition system)
Local variables P : an abstract property-game, initialized with $P^I_{T,A}$

1: Simplify (G)
2: `while` ($\omega(c^{\mathrm{i}}) = \bot$) `do`
3: Redirect every transition t pointing to a hyper-point $(\widetilde{Z}, q) \in \mathbb{P}(Z) \times Q_{\mathrm{qua}}$ such that it points to $(\pi_1(\{(\widetilde{Z}, q)\}.R^-), q)$, where this hyper point together with outgoing junction transitions to $\{(\widetilde{Z}, q)\}.R^-$ are added to C (for example by using procedure Add).
% This step updates $\widetilde{Z}$ 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 $\omega(c') = \bot$ and in addition, either $\delta(q') = \Diamond q \wedge \omega(c) = \mathrm{tt}$, or $\delta(q') = \Box q \wedge \omega(c) = \mathrm{ff}$. % Game-states that have no influence on (in)validity are removed. States fullfilling 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 $\overline{\mathcal{L}}$
6: Simplify (G)
7: `od`
8: `return` $\omega(c^{\mathrm{i}})$

Table 4. A model checking algorithm for μ-calculus properties, where refinement is performed 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.

Algorithm Refine (P: an abstract property-game, $((z, q), p)$: $(Z \times Q) \times \overline{\mathcal{L}}$)
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 $[\![\varrho(z_1)]\!] = [\![\varrho(z) \wedge p]\!]$ and $[\![\varrho(z_2)]\!] = [\![\varrho(z) \wedge \neg p]\!]$
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 the game-states obtained from splitting.
5: `if` $c^{\mathrm{i}} = (z, \tilde{q})$ `then (if` Satisfiable $(p^{\mathrm{i}} \wedge \varrho[z_1])$ `then` $c^{\mathrm{i}} := (z_1, \tilde{q})$ `else` $c^{\mathrm{i}} := (z_2, \tilde{q}))$ % Relocation of the initial game-state.
6: `if` $\tilde{q} \notin Q_{\mathrm{qua}}$ `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: `od`
11: `else` % $\tilde{q} \in Q_{\mathrm{qua}}$
12: OutgoingMayCalculation $(P, z, z_1, z_2, \tilde{q})$
13: OutgoingMustCalculation $(P, z, z_1, z_2, \tilde{q})$
14: `fi`
15: `while` $(R^-.\{(z, \tilde{q})\}) \cap (Z \times Q) \neq \{\}$ `do` % Calculation of the incoming junction transitions
16: remove an element $(z', \tilde{q}')$ from $(R^-.\{(z, \tilde{q})\}) \cap (Z \times Q)$ % By an invariant $z' = z$
17: $Q' := Q' \cup \{\tilde{q}'\} \setminus \{\tilde{q}\}$; $R^- := R^- \cup \{((z_1, \tilde{q}'), (z_1, \tilde{q})), ((z_2, \tilde{q}'), (z_2, \tilde{q}))\}$
18: `od`
19: IncomingMayCalculation $(P, z, z_1, z_2, \tilde{q})$
20: IncomingMustCalculation $(P, z, z_1, z_2, \tilde{q})$
21: $C := C \setminus \{(z, \tilde{q})\}$; $C_1 := C_1 \setminus \{(z, \tilde{q})\}$; $C_2 := C_2 \setminus \{(z, \tilde{q})\}$
22: `od`

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

target states (Line 9). In case that $\tilde{q} \in Q_{\text{qua}}$ 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 $\widetilde{Z}$, 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 $(\widetilde{Z}, \tilde{q})$ is 'added' if z_i and $\mathcal{U}_G(\tilde{q}) \cup \widetilde{Z}$ 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 $\tilde{q} \in Q_{\text{qua}}$,

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

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 $\widetilde{Z}$ are considered that are subsets of the targets of the may transitions leaving the corresponding new game-state. Furthermore, $\widetilde{Z}$ is automatically not considered if a superset is already determined to not fulfill the must condition. Similarly, once $\widetilde{Z}$ is determined to be a hypertransition, none of its supersets is checked. This is justified by the fact that including only *minimal* sets $\widetilde{Z}$ as hyper-points does not damage precision [36].

Consider now the incoming transitions (Lines 15-20). 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 17). 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 $(\widetilde{Z}', \tilde{q}')$. We consider such must transitions as incoming must transitions. The hyper-point $\widetilde{Z}'$ 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 $\widetilde{Z}'$ 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 the function 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 illustrated in Figure 2 (e), where a refinement heuristic determined that the game-state $c = (\ell = 1 \wedge x \neq 1, \Box)$ 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 $(\ell = 1 \wedge x \in \{0, 2\}, \Box)$ and $(\ell = 1 \wedge x > 2, \Box)$. 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 $(\ell = 1 \wedge x \in \{0, 2\}, \Box)$, while the second remains as an outgoing may transition of $(\ell = 1 \wedge x > 2, \Box)$. The latter transition is also added as an outgoing must transition of $(\ell = 1 \wedge x > 2, \Box)$,

as it now fulfills the must condition (more precisely, a hyper-point $(\{\ell = 1 \wedge x \neq 1\}, \Box)$ is added, with an incoming must transition from $(\ell = 1 \wedge x > 2, \Box)$, 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 $(\ell = 1 \wedge x > 2, \Box)$, 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. However, our algorithm is sound as well as relatively complete for least fixpoint-free μ-calculus formulas:

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

Proof. Consider the following invariant of the abstract property-games which arise in PropertyCheck (A,T):

- All abstract property-games have the following properties: (i) the abstract state in the target of a transition can differ from the abstract state in its source game-state only if the property of the source game-state is from Q_{qua} and (ii) the property of a game-state can be changed along a transition only as it changes in A,
- a concrete transition leaving a state in $[\![\varrho(z)]\!]$ can either (i) be matched at any game-state (z, q), where $q \in Q_{\text{qua}}$, by a corresponding may transition (such that the target abstract state and the concrete target are related via ϱ) or (ii) the corresponding target of such a hypothetic may transition is valid, resp. depending on the type of the property invalid, (i.e., no winning strategy would use this transition),
- if a must hypertransition from an abstract state (z, q) to (Z, q) exists then for every concrete state $s \in [\![\varrho(z)]\!]$ there is a transition into a concrete state related via ϱ to (i) an element from Z or to (ii) an abstract state z' such that $(z', \text{succ}(q))$ is valid, resp. depending on the type of the property invalid,
- if a transition from a hyper-point or from a game-state where the property is of type $\tilde{\wedge}$ or $\tilde{\vee}$ is missing then the corresponding target is valid, resp. depending on the type of the property invalid, and
- if a game-state (z, q) becomes valid (invalid) then all concrete states in $[\![\varrho(z)]\!]$ satisfy (resp. falsify) q.

This invariant holds for the initial abstract property-game trivially, and is maintained by the different steps of the algorithm. To verify the first four items more detailed invariants have to be formulated regarding Refine and its used procedures. As for the last item, the first four items ensure that a winning strategy for Player 1 (Player 2) in the validity (resp. invalidity) game over the abstract property-game starting at (z, q) can be mapped to a winning strategy for Player 1 (resp. Player 2) in the validity (resp. invalidity) game over the concrete property-game starting at (s, q) for every $s \in [\![\varrho(z)]\!]$. This means that every concrete state $s \in [\![\varrho(z)]\!]$ satisfies (resp. falsifies) q, which proves the last item. The statement of the theorem follows from the last item immediately. □

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* $T \models A$*, then there exists a (not necessarily computable) refinement heuristic* Heuristic *such that* PropertyCheck(A, T) *returns* tt*.*

Proof. In [10] the authors present a partition of the concrete state space (together with ranking functions and a set J) that yields completeness for the full μ-calculus, i.e., if the concrete system satisfies the formula then the corresponding abstract model satisfies it as well. Since we refer to formulas that do not contain least fixpoints, no ranking functions are necessary. Furthermore, since J only has influence on the fairness constraint, which does not occur here, it can be chosen such that the resulting abstract model yields a generalized Kripke modal transition system (note that this is not true in the general case).

Now consider a refinement heuristic that splits every game-state according to the above partition until the configuration structure which is based on the above generalized Kripke modal transition system is reached. In fact, an abbreviation of the configuration structure will be reached (since transitions and states can be removed in our algorithm during simplification). From the fact that only transitions and states that are irrelevant are removed and from the fact that the hypertransitions calculation also takes into account the valid, resp. invalid, game-states, it follows that the obtained configuration structure also satisfies the property. □

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., [9]. 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 [7,10], 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 $\omega(z,q) = \bot$. 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 ($\mathrm{sor}(t) = \mathrm{sor}(t')$) has a target $\mathrm{tar}(t')$ whose first component $\widetilde{Z}$ is different from the singleton set consisting of the first component z of the target of t (i.e., $\pi_1(\mathrm{tar}(t')) \neq \{\pi_1(\mathrm{tar}(t))\}$).

A refinement heuristic $\mathsf{Heuristic}$ 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 $\mathsf{Heuristic}(P) = (c,p)$ then (i) $c = (z,q)$ is a predicate-unknown state in C and $p = \delta(q)$ or (ii) $c = \mathrm{sor}(t)$ for some real may transition $t \in R^+$ and $p = \mathrm{pre}(\varrho(\pi_1(\mathrm{tar}(t))))$ or (iii) neither a predicate-unknown state nor a real may transition exists.

A predicate-unknown state or a real may transition always exists in simplified abstract property-games:

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

Proof. The proof is by contraposition. If no predicate-unknown state or real may transition exists, then every winning strategy for Player 1 in the invalidity game, which refutes invalidity, is also a winning strategy in the validity game, which establishes validity, and vice versa. Hence, the result is a two valued approach. Thus, the abstract property-game is (in)valid in c^{i}. In particular, if the abstract property-game is simplified, then $\omega(c^{\mathrm{i}}) \neq \bot$. Contradiction. □

Intuitively, predicate-unknown states and real may transitions are good candidates for refinement since they can be viewed as a cause for uncertainty. In particular, the refinement heuristics used in the example of Section 2 are all pre-based. Pre-based refinement heuristics are sufficient for finite state systems (but often also works for infinite 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* $\mathsf{Heuristic}$ *is a pre-based refinement heuristic. Then* $\mathsf{PropertyCheck}(A,T)$ *terminates, i.e., returns* tt *or* ff.

Proof. Since every iteration of the algorithm is finite, nontermination can only occur if infinitely many refinements are made. We show that this is impossible. The following facts can be easily checked. (i) Every set of concrete states described by an abstract state is closed under bisimilarity (since an abstract state is never split in a way that separates bisimilar concrete states). (ii) An abstract state that has only one related bisimilarity-equivalence class of concrete states cannot be predicate-unknown and can only have may transitions that are also must transitions. (iii) Every resolving of an abstract state via a pre-based refinement heuristic yields a non-trivial division of the related concrete states. Hence, only finitely many such refinement steps can be made, since only finitely many bisimilarity-equivalence classes exist (and Q is finite). □

5.1 Bottom-up strategy

Determine (i) a predicate-unknown state (z,q) or (ii) a real may transition t pointing to an (in)valid game-state, i.e., $\omega(\mathrm{tar}(t)) \neq \bot$. Return (i) the predicate-unknown state with the corresponding predicate as split information, i.e., $((z,q),\delta(q))$, resp. (ii) the source of t combined with the weakest precondition of the target of t, i.e., $(\mathrm{sor}(t), \mathrm{pre}(\varrho(\pi_1(\mathrm{tar}(t)))))$. 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 $\mathrm{pre}(\varrho(\ell = 1 \wedge x = 1)) = (\ell = 1 \wedge x = 2) \vee (\ell = 0 \wedge x = 0)$, 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 the one from Figure 2 (e), which is already equivalent to the one from Figure 2 (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 $\ell = 1 \wedge x \neq 1 \wedge \ldots \wedge x \neq i$, 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. $(\text{sor}(t), \text{pre}(\varrho(\pi_1(\text{tar}(t)))))$. If more than one such state exists, an optimization is to favor predicate-unknown states and real may transitions pointing to (in)valid game-states, by the same argument presented for the bottom-up strategy. 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.

A disadvantage of the breadth first strategy is that the state-space usually increases faster than with a bottom-up strategy. This is because the split is performed "higher" in the game structure. Therefore, when it is more crucial to keep the state space small, e.g. when memory problems occur or when the validity check on a great state-space will outweigh the refinement calculation time, the bottom-up strategy should be favored. Still, in some cases the breadth first strategy reduces the number of unnecessary refinement steps compared to the bottom-up strategy, since the split is performed closer to the initial game-state. Thus, the influence of the split state on the value of the initial game-state is sometimes more substantial. It is even possible that the algorithm terminates with the breadth first strategy, where it fails to terminate with the bottom-up strategy, as demonstrated by Example 3 below. Nevertheless, it is still possible that the bottom-up strategy needs less refinement steps, if the bottom-up strategy backwardly determines a path/tree that indicates (in)validity.

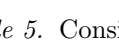
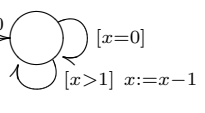
Example 3. Consider the example from Section 2. Then the breadth first strategy will split one of the $(true, \diamond)$ states in Figure 2 (a) along the weakest precondition of *true*, as it is made in Figure 2 (f). Thus the property will be shown after a single refinement step.
The success of the breadth first strategy in this example is due to the shallow depth of the loop $q_0 \to q_5 \to q_0$, 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 $q_0 \to q_5 \to q_0$ is replaced by a 'deeper' loop $q_0 \to q_5 \to q_6 \to \ldots \to q_n \to q_0$ in which $q_6, \ldots, q_n$ 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.

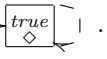
5.3 Youngest first strategy

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 $(true, \ell = 1)$. 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:

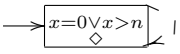
Example 5. Consider the property  checked on the system 

, where $x \in \mathbb{N}$. The initial abstract property-game is 

. In addition,

$\text{pre}(true) = x = 0 \vee x > 1,$
$\text{pre}(x = 0 \vee x > 1) = x = 0 \vee x > 2,$
$\text{pre}(x = 0 \vee x > 2) = x = 0 \vee x > 3, \ldots$

Thus any pre-based refinement heuristic will produce after n refinements the (simplified) abstract property-game



i.e., the property cannot be verified. On the other hand, if the initial state is separated first, then we obtain

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 a backward search when a state is determined to become (in)valid as a result of the split, and Simplify is only called for more exact determination of least fixpoint properties. In other words, inexpensive, but imprecise, validity determination (especially those obtained by construction) are to be applied regularly; the expensive parity game algorithm, on the other hand, is to be applied seldom for obtaining precision.

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. Hence, 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. 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. Here, a tradeoff between time and space arises. Space optimizations can be used as in [23].

Unnecessary theorem prover calls. In case of refinement, a forward search that propagates the split to the game-states reachable from the currently split state via junction transitions always takes place. In particular, the split is sometimes propagated to $\Diamond$- and $\Box$-states, where the outgoing may and must transitions need to be recalculated. 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. Consequently, some calculations via theorem prover calls of the outgoing may and must transitions can be avoided.

Furthermore, the absence of a may transition or the existence of a must (hyper)transition can be determined only by demand by initially adding all candidates as uncertain may, resp. must, transitions and letting the heuristic determine if and when to check if they are real may, resp. real must, transitions. This has the advantage that theorem prover calls for checking the existence of may or must transitions can be avoided if one expects that they are not needed in order to verify the property. If such an approach is used, calls of Simplify should definitely be reduced (see first optimization), since only a few improvements will take place during a single refinement step.

Complex pre(ψ)-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 pre(ψ) 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 system (e.g., by partitioning the code-lines). The techniques of interpolations [20,25,32] might also help to avoid the high cost of pre(ψ)-calculations.

Of course, the common approach of splitting along all the atomic predicates that occur in the weakest precondition formula ϕ, rather than splitting along ϕ itself, can be applied in order to avoid the exact calculation of ϕ and thus simplify the theorem prover calls. The price to be paid is the generation of an increased number of states during a refinement step.

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 compromise precision and use further approximations. In this case, 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 the 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 [37].
- Approximate the system, e.g., instead of using T, use an approximated system for which pre(ψ) can be more efficiently calculated. In particular, side effect free executions can be subsumed into atomic executions.

– Approximate the theorem prover queries by clustering predicates [4,24]. 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 (mainly disproving) safety properties, since by using a 3-valued abstract model better refinement heuristics can be obtained. These directions are investigated further in our more recent work [13], where a 9-valued logic is used and the refinement is made even more local by splitting a *single* configuration (game-state) at each refinement step, enabling the definition of more refinement heuristics. 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.

A Pseudo code of additional procedures

In this section, the pseudo code of the procedures used in Refine are presented: Add is presented in Table 6, OutgoingMayCalculation and OutgoingMustCalculation are presented in Table 7, and IncomingMayCalculation and IncomingMustCalculation are presented in Table 8. Their informal description is given in Section 4 and inside the algorithms. Satisfiable corresponds to a satisfiability check made by a theorem prover and is not discussed further in this paper.

Note that if the abstraction is precise then in OutgoingMayCalculation and IncomingMayCalculation the second satisfiability check, that checks the may condition with respect to z_2, must be successful, and thus need not be checked in case that the first one (that checks the may condition with respect to z_1) is not successful. This is because the precision of the abstraction ensures that the may condition holds for z, which represents the union of z_1 and z_2. Thus, if the may condition does not hold for z_1 it must hold for z_2. As a result, the second satisfiability check can be skipped in this case. This simplification should only be used if the initial abstraction is precise as well (i.e., all the may transitions fulfill the may condition and no redundant may transitions are included), otherwise the relative completeness is lost (soundness remains guaranteed in both approaches).

Acknowledgements. This work is in part financially supported by the DFG projects FE 942/1-1 and FE 942/2-1.

References

1. T. Ball and O. Kupferman. An abstraction-refinement framework for multi-agent systems. In *LICS*, pages 379–388. IEEE Computer Society Press, 2006.
2. T. Ball, O. Kupferman, and M. Sagiv. Leaping loops in the presence of abstraction. In *CAV*, volume 4590 of *LNCS*, pages 491–503. Springer, 2007.
3. 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.
4. E. Clarke, H. Jain, and D. Kroening. Predicate abstraction and refinement techniques for verifying verilog. Technical Report CMU-CS-04-139, 2004.
5. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.
6. 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.
7. 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.
8. L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *LICS*, pages 170–179. IEEE Computer Society Press, 2004.
9. H. Fecher and M. Huth. Complete abstraction through extensions of disjunctive modal transition systems. Technical Report 0604, Christian-Albrechts-Universität zu Kiel, 2006.
10. 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.
11. H. Fecher and M. Huth. Model checking for action abstraction. In *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2008.
12. H. Fecher and S. Shoham. Local abstraction-refinement for the mu-calculus. In *SPIN*, volume 4595 of *LNCS*, pages 4–23. Springer, 2007.
13. H. Fecher and S. Shoham. State focusing: Lazy abstraction for the mu-calculus. In *SPIN*, volume 5156 of *LNCS*, pages 95–113. Springer, 2008.
14. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, volume 2154 of *LNCS*, pages 426–440. Springer, 2001.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
16. 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.

Algorithm Add (P: an abstract property-game, (η, q): $(Z \times Q) \cup (\mathbb{P}(Z) \times Q_{\text{qua}})$)

1: **if** $(\eta, q) \notin C$ **then** % game-state is only added if it is not yet present
2: $C := C \cup \{(\eta, q)\}$ % the game-state is added and attributed to the corresponding player in the succeeding two lines
3: **if** $(\tilde{q} \in Q_1 \wedge \eta \in Z) \vee (\tilde{q} \in Q_2 \wedge \eta \in \mathbb{P}(Z))$ **then** $C_1 := C_1 \cup \{(\eta, q)\}$
4: **if** $(\tilde{q} \in Q_2 \wedge \eta \in Z) \vee (\tilde{q} \in Q_1 \wedge \eta \in \mathbb{P}(Z))$ **then** $C_2 := C_2 \cup \{(\eta, q)\}$
5: $\theta(\eta, q) := \Theta(q)$ % the parity value is derived from its property
6: $\omega(\eta, q) := \bot$ % its validity is set to 'unknown' except if the property corresponds to a predicate, in which case its validity is determined by satisfiability checks as done in the next lines
7: **if** $\delta(q) \in \mathcal{L}$ **then** % in this case $\eta \in Z$
8: **if** $\neg$ Satisfiable $(\varrho[\eta] \wedge \neg\delta(q))$ **then** $\omega(\eta, q) := \text{tt}$
9: **else if** $\neg$ Satisfiable $(\varrho[\eta] \wedge \delta(q))$ **then** $\omega(\eta, q) := \text{ff}$
10: **fi**
11: **if** $\eta \in \mathbb{P}(Z)$ **then** $R^- := R^- \cup \{((\eta, q), (z, \text{succ}(q))) \mid z \in \eta\}$ % if it is a hyper-point, its derived outgoing transitions are added
12: **fi**

Table 6. Procedure that adds a new game-state.

Algorithm OutgoingMayCalculation $(P, z, z_1, z_2, \tilde{q})$ % calculation of the may transitions leaving $(z_1, \tilde{q})$ or $(z_2, \tilde{q})$, which is done by replacing every may transition outgoing the to be spit game-state by corresponding may transitions outgoing the split game-states, if they fulfill the may condition respectively.

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 already determined relevant must transitions for $(z_i, \tilde{q})$; N_i stores the not yet considered, relevant must transitions for $(z_i, \tilde{q})$

1: $M_1 = \{\widetilde{Z} \cap \pi_1(\{(z_1, \tilde{q})\}.R^+) \mid \widetilde{Z} \in \pi_1(\{(z, \tilde{q})\}.R^-)\}$; $M_2 = \{\widetilde{Z} \cap \pi_1(\{(z_2, \tilde{q})\}.R^+) \mid \widetilde{Z} \in \pi_1(\{(z, \tilde{q})\}.R^-)\}$ % M_i is initialized with the must transitions of the to be spit state, except that the target sets are restricted to the corresponding possible may targets, since precision is increased by removing irrelevant targets
2: remove all elements from $\{(z, \tilde{q})\}.R^-$ % their useful information is already encoded in M_i
3: $N_1 := \pi_1(\mathbb{P}(\{(z_1, \tilde{q})\}.R^+)) \setminus \{\widetilde{Z}_1 \mid \exists \widetilde{Z} \in M_1 : \widetilde{Z} \subseteq \widetilde{Z}_1\}$; $N_2 := \pi_1(\mathbb{P}(\{(z_2, \tilde{q})\}.R^+)) \setminus \{\widetilde{Z}_2 \mid \exists \widetilde{Z} \in M_2 : \widetilde{Z} \subseteq \widetilde{Z}_2\}$ % N_i is initialized with all the sets being a subset of the corresponding may targets such that no must transition which subsumes it is guaranteed, i.e. no subset of it exists in M_i
4: **for i=1 to 2 do**
5: **While** $N_i \neq \{\}$ **do**
6: take (not remove) an element $\widetilde{Z}$ from N_i
7: **if** $\neg$ (Satisfiable $(\varrho(z_i) \wedge \neg(\text{pre}(\bigvee_{\tilde{z} \in \mathcal{U}_G(\tilde{q}) \cup \widetilde{Z}} \varrho(\tilde{z}))))$) **then** % check if a must transition exists (iff $\varrho(z_i) \Rightarrow \text{pre}(\bigvee_{\tilde{z} \in \mathcal{U}_G(\tilde{q}) \cup \widetilde{Z}} \varrho(\tilde{z}))$) by using a satisfiability check. Here, the targets of the transitions that are removed through optimizations also have to be taken into account, which is handled by $\mathcal{U}_G(\tilde{q})$.
8: $M_i := M_i \cup \{\widetilde{Z}\} \setminus \{\widetilde{Z}_i \mid \widetilde{Z} \subset \widetilde{Z}_i\}$; $N_i := N_i \setminus \{\widetilde{Z}_i \mid \widetilde{Z} \subseteq \widetilde{Z}_i\}$ % if $\widetilde{Z}$ is the target of a must transition, it is added to M_i and its supersets are removed from M_i, since $\widetilde{Z}$ is a more precise witness. Furthermore, $\widetilde{Z}$ and its supersets are removed from N_i.
9: **else** $N_i := N_i \setminus \{\widetilde{Z}_i \mid \widetilde{Z}_i \subseteq \widetilde{Z}\}$ % if $\widetilde{Z}$ is not the target of a must transition, it is removed from N_i along with its subsets, since its subsets also cannot be must transition targets.
10: **od**
11: **For** $\widetilde{Z} \in M_i$ **do** % Add the calculated hyper-transitions, including the necessary hyper-points
12: Add $(P, (\widetilde{Z}, \tilde{q}))$; $R^- := R^- \cup \{((z_i, \tilde{q}), (\widetilde{Z}, \tilde{q}))\}$
13: **next**
14: **next**

Table 7. Procedures for the calculation of the outgoing may, resp. must, transitions.

Algorithm IncomingMayCalculation $(P,\ z,\ z_1,\ z_2,\ \tilde{q})$ % calculation of the weak transitions incoming $(z_1, \tilde{q})$ and $(z_2, \tilde{q})$, which is done by replacing every may transition incoming the to be spit game-state by corresponding may transitions incoming the split game-states, if they fulfill the may condition respectively.

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

Algorithm IncomingMustCalculation $(P,\ z,\ z_1,\ z_2,\ \tilde{q})$ % calculation of the must transitions incoming $(z_1, \tilde{q})$ and $(z_2, \tilde{q})$
Local variables $x : \{0,1\}$ % $x = 1$ indicates that the hypertransition where z is replaced by both z_1 and z_2 becomes redundant

1: `While` $R^-.\{(z,\tilde{q})\} \neq \{\}$ `do` % hyper-points that point to the to be split game-state are transformed to hyper-points where z is replaced by the split game-states z_1 or z_2 or both, depending on the source of the corresponding hypertransitions, as described below.
2: remove an element $(\widetilde{Z}', \tilde{q}')$ from $R^-.\{(z,\tilde{q})\}$ % By an invariant $z \in \widetilde{Z}'$
3: `While` $R^-.\{(\widetilde{Z}', \tilde{q}')\} \neq \{\}$
4: remove an element $(\tilde{z}', \tilde{q}')$ from $R^-.\{(\widetilde{Z}', \tilde{q}')\}$ % a source state of a corresponding hypertransition is chosen
5: $x := 0$ % initialize x for every loop iteration
6: `For` $i = 1$ `to` 2 `do`
7: `if` $\neg$ `(Satisfiable` $(\varrho(\tilde{z}') \wedge \neg(\mathrm{pre}(\bigvee_{\tilde{z} \in \mathcal{U}_G(\tilde{q}') \cup \{z_i\} \cup \widetilde{Z}' \setminus \{z\}} \varrho(\tilde{z}))))$ `then` % check if a must transition exists, where z is replaced by z_i. Here, the targets of the transitions that are removed through optimizations also have to be taken into account, which is handled by $\mathcal{U}_G(\tilde{q})$.
8: Add $(P, \{z_i\} \cup \widetilde{Z}' \setminus \{z\}, \tilde{q}')$; $R^- := R^- \cup \{((\tilde{z}', \tilde{q}'), (\{z_i\} \cup \widetilde{Z}' \setminus \{z\}, \tilde{q}'))\}$; $x := 1$ % if the must transition exists, the corresponding hyper-point and transitions are added and x is set to 1, since the hypertransition where z is replaced by both z_1 and z_2 is redundant (a more precise, i.e. lower, must hypertransition already exists)
9: `fi`
10: `next`
11: `if` $x = 0$ `then` % if no lower hypertransition is derived, the one where z is replaced by both z_1 and z_2 is added
12: Add $P, (\{z_1, z_2\} \cup \widetilde{Z}' \setminus \{z\}, \tilde{q}')$; $R^- := R^- \cup \{((\tilde{z}', \tilde{q}'), (\{z_1, z_2\} \cup \widetilde{Z}' \setminus \{z\}, \tilde{q}'))\}$
13: `fi`
14: `od`
15: remove all elements from $\{(\widetilde{Z}', \tilde{q}')\}.R^-$; $C := C \setminus \{(\widetilde{Z}', \tilde{q}')\}$; $C_1 := C_1 \setminus \{(\widetilde{Z}', \tilde{q}')\}$; $C_2 := C_2 \setminus \{(\widetilde{Z}', \tilde{q}')\}$ % the hyper-point containing the to be split game-state is removed.
16: `od`

Table 8. Procedures for the calculation of the incoming may, resp. must, transitions.

17. 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*, 205(8):1130–1148, 2007.
18. B. Gulavani, T. A. Henzinger, Y. Kannan, A. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *FSE*. ACM, 2006.
19. A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In *TACAS*, volume 3920 of *LNCS*, pages 212–226. Springer, 2006.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
22. 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. Springer, 2001.
23. H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, volume 3440 of *LNCS*, pages 397–412. Springer, 2005.
24. 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.
25. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.
26. M. Jurdzinski. Deciding the winner in parity games is in UP $\cap$ co-UP. *Inf. Process. Lett.*, 68(3):119–124, 1998.
27. 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.
28. D. Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

29. R. Küsters. Memoryless determinacy of parity games. In *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*, pages 95–106. Springer, 2002.
30. K. G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210. IEEE Computer Society Press, 1988.
31. K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *LICS*, pages 108–117. IEEE Computer Society Press, 1990.
32. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
33. K. S. Namjoshi. Abstraction for branching time properties. In *CAV*, volume 2725 of *LNCS*, pages 288–300. Springer, 2003.
34. A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *DAC*, pages 457–462, 1998.
35. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.
36. S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004.
37. S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. In *LICS*, pages 399–410. IEEE Computer Society Press, 2006.
38. Th. Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Soc. Math. Belg.*, 8(2):359–391, May 2001.
39. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.