Christian-Albrechts-University of Kiel
Institute of Computer Science
Chair of Software Technology

Diploma Thesis

Using generalized μ-automata for local abstraction-refinement

Jan Waller

July 23, 2008

Advised by:
Prof. Dr. Willem-Paul de Roever
Dr. Harald Fecher
Heiko Schmidt

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel, den

Contents

1 Introduction **1**
1.1 Contribution . . . 2
1.2 Related work . . . 2
1.3 Outline . . . 3

2 Preliminaries **4**
2.1 Finite State Machines . . . 4
2.2 Pointed Kripke Structures . . . 6
2.3 Alternating Tree Automata . . . 9
2.4 Parity Games . . . 11

3 Underlying Structures and Methods **16**
3.1 Predicate Abstraction . . . 16
3.2 Abstraction Techniques . . . 17
3.3 Generalized μ-Automata . . . 21
3.4 Abstract Property Games . . . 30

4 CEGAR **37**
4.1 Counter-Example Guided Abstraction Refinement . . . 37
4.2 Simplified Games . . . 38
4.3 Property Check . . . 40
4.4 Refinement . . . 41
4.5 Soundness of the Algorithm PropertyCheck . . . 51
4.6 Heuristics . . . 54
4.7 Optimization . . . 56
4.8 Example . . . 57

5 Conclusion **65**
5.1 Future Work . . . 65

Algorithms **66**

List of Figures **72**

Bibliography **73**

Chapter 1

Introduction

Model checking has its origins in the early 1980s, when Clarke and Emerson [2] as well as Queille and Sifakis [17] introduced a new algorithmic approach for the verification of systems, called model checking. One formally defines satisfaction as a relation $M \models \phi$, where M is a mathematical model of a system and ϕ is a property, which has to be shown, encoded within a formal language.

A possible formal language to encode properties is the modal μ-calculus [15]. The μ-calculus allows the expression of safety and reachability properties, as well as combinations of these properties. Furthermore, the modal μ-calculus can express most modal and temporal logics, such as linear temporal logic (LTL) and computational tree logic (CTL and CTL*), as shown by Emerson [5].

The mathematical model of the system is usually encoded as a pointed Kripke structure. These Kripke structures often have a large or even infinite state space. Thus instead of directly calculating $M \models \phi$, one employs abstraction techniques, i.e. one constructs a smaller abstract model A from the specification of M such that $A \models \phi$ implies $M \models \phi$. Thus to counter the state space explosion, predicate abstraction, as defined by Graf and Saidi [10], is employed.

A successful technique to automatically construct such an abstract model A from a given large concrete model M is *counterexample-guided abstraction refinement* (CEGAR) developed by Clarke et al. [3]. The CEGAR approach typically consists of three steps: construction of an abstract model, model checking against the property, and refinement of the abstract model to get a new abstract model. These steps are repeated until the generated abstract model is sufficient to prove or disprove the property. The usual approach is to split the abstract states contained in a spurious counter example and construct a new abstract model. Another approach is lazy-abstraction, implemented in BLAST, a prominent tool based on CEGAR, developed by Henzinger et al. [14]. There, the refinement is performed directly on the abstract model to avoid rebuilding of the abstract model after each step.

An abstraction technique is a way to describe such an abstract model. One such abstraction technique was developed by de Alfaro et al. [4] and later transformed by Fecher and Huth [6], who called it *pre-abstraction*. This technique utilizes generalized Kripke modal transition systems as abstract models. Additionally, Fecher and Huth [6] developed a new abstraction technique called *post-abstraction*, utilizing μ-automata as abstract models. Depending on the concrete model M and the proposition ϕ, the calculation of pre- or post-abstraction may be preferable to each other. In addition they

sketched how a combination of both techniques may result in a more precise abstraction without increasing the cost of the abstraction synthesis too much.

1.1 Contribution

In this diploma thesis, I specify the combination of pre- and post-abstraction, sketched in [6]. This new abstraction technique utilizes generalized μ-automata, first used in my student research project [18]. Based upon this abstraction technique I present a lazy CEGAR algorithm, which verifies a given model against a property.

This algorithm is based upon *three-valued satisfaction games*. A verifier tries to obtain validity, a falsifier tries to obtain invalidity, and a third value (unknown, $\perp$) captures the possibility that no player wins. So, the model checking is performed via a reduction of the three-valued satisfaction game into two games: a validity-game and an invalidity-game. Thus, contrary to traditional CEGAR approaches, unknown results rather than invalid results lead to refinement.

Furthermore, the algorithm uses a stronger notion of laziness. Only some configurations of the satisfaction game containing a single abstract state are split at each refinement step, thus the state space remains small. Additionally, since the algorithm operates upon a satisfaction game instead of the abstract model, the game can be simplified after each refinement step. Finally, the algorithm allows for a heuristic to determine the next local refinement step.

1.2 Related work

In my student research project [18] I worked on a combination of pre- and post-abstraction for the first time. This resulted in the abstract model of generalized μ-automata. In addition I gave a sound definition of satisfaction with the help of satisfaction games between this new model and alternating tree automata. Furthermore, I gave a definition of refinement with the help of refinement games between two generalized μ-automata, which is reflexive, transitive and sound.

Grumberg et al. [11, 12] developed a CEGAR based algorithm for the verification of properties encoded in the μ-calculus utilizing three-valued satisfaction games. Fecher and Shoham [7] developed a new CEGAR based model checking algorithm based upon [11, 12]. There, they utilized generalized Kripke modal transition systems [6] and the technique of pre-abstraction for an algorithm similar to the one presented here. In [8], Fecher and Shoham developed a new technique named state focusing as a replacement for state splitting. This new technique allows for an even lazier algorithm than [7] without any of its drawbacks, like the possibility of state explosion.

Pardo and Hachtel [16] gave a CEGAR-approach to branching time properties, where the state space remains unchanged and only the set of transition relations is under and respectively over approximated.

The tool SLAM [1] deals with the state space doubling problem, that occurs after an algorithm splits the whole state space via a predicate, with the help of BDDs. This

approach is generalized from safety properties checked by the tool to μ-calculus properties in the tool YASM by Gurfinkel and Chechik [13], using a equivalent of Kripke modal transitions systems as the underlying abstract structure, which is less expressive than generalized Kripke modal transitions systems used by pre-abstraction.

1.3 Outline

In the second chapter, pointed Kripke structures and alternating tree automata are briefly introduced. Furthermore, three-valued satisfaction games (*strong-weak-parity-games*) and satisfaction for pointed Kripke structures (*property games*) are presented. Chapter 3 introduces abstraction and generalized μ-automata, as well as property games for them. Additionally, a notion of refinement between two generalized μ-automata is presented. Finally, the third chapter concludes with the introduction of abstract property games, which are used by the algorithm presented in chapter 4.

Chapter 2

Preliminaries

In the following, $\mathbb{P}(S)$ denotes the power set of a set S. Let $\mathbb{N}$ denote the natural numbers, while $\mathbb{N}_0$ denotes the natural numbers including zero. Let 0 be even. For a relation $\rho \subseteq B \times C$ with $X \subseteq B$ I will write $X.\rho$ for $\{c \in C \mid \exists b \in X : (b, c) \in \rho\}$ and with $b \in B$ I will write $b.\rho$ for $\{c \in C \mid (b, c) \in \rho\}$. Let $\dot{\cup}$ denote the disjoint union of sets. Let $\Pi_i(X)$ denote the projection to the ith component of the tuple X.

2.1 Finite State Machines

A finite state machine is a model of the behavior of a system. Basically, a finite state machine is a directed graph. Vertices of the graph represent states, while arcs represent transitions. Additionally a number of variables is assigned to a finite state machine.

Each state stores the current assignment of the variables. Each transition, leading from one state to another (or to the same), has a label, consisting of the *guard* and the *action*. A transition can be taken, if the guard with the current assignment of variables is valid. If a transition is taken, the assignment of variables at the next state is changed according to the action.

Definition 2.1.1 (Finite state machine)**.** A *finite state machine* M is defined as the tuple $M = (S, \hat{s}, V, \hat{v}, \rightarrow)$ with

- S is the finite set of states,
- $\hat{s} \in S$ is the initial state,
- V is the set of Variables,
- $\hat{v}$ is the initial assignment of variables, and
- $\rightarrow \subseteq S \times \text{GUARDS} \times \text{ACTIONS} \times S$ is the set of transitions, where
  - GUARDS is the set of guards and
  - ACTIONS is the set of actions.

Each guard of GUARDS is usually a formula utilizing variables in V and simple operators. Each action of ACTIONS is usually a assignment of variables, often depending on the prior assignment.

The calculation of the weakest pre-condition of any assignment of values to the set of variables can efficiently be performed. To calculate the pre-condition of a variable

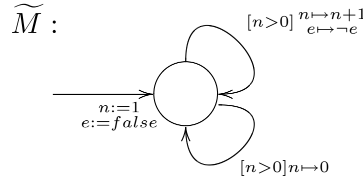

Figure 2.1: A *finite state machine*. The range of n is $\mathbb{N}_0$, initialized with 1. The range of e is boolean and initialized with $false$. The actions of the transitions can be executed whenever the guard, depicted in rectangular brackets, evaluates to true.

assignment v for any transition $t = (s_1, g, a, s_2)$, one simply calculates $\text{pre}_t(v) = g \wedge v[a]$, where $[\cdot]$ stand for the textual replacement. So for $v = (n = 0)$ and $a = n \mapsto n + 1$ one gets $v[a] = (n = 0)[n \mapsto n + 1] = (n + 1 = 0) = (n = -1)$.

To calculate the pre-condition of a variable assignment v for the whole finite state machine, one calculates the disjunction of the pre-conditions over all transitions. If there are several states, one has to adapt the guards and actions to include the changes between the states. An example can be seen in Example 2.1.2.

Example 2.1.1. Consider a simple toy example: a program to determine if a number ($\in \mathbb{N}$) is even or odd. The idea is to start with 1, which is odd, and increase by one, switching from odd to even or from even to odd with every increment step. At each step, the program non-deterministically chooses either to continue to the next higher number or to finish, which allows e.g. the printing of the status of the current number.

A possible finite state machine $\widetilde{M}$ to describe such a system is illustrated in Figure 2.1. It consists of a single state and two variables, $n \in \mathbb{N}_0$ and $e \in \{true, false\}$. The machine is initialized with $n = 1$ and $e = false$, i.e. 1 is an odd number. As long as n is greater than zero there are always two possible transitions to take. One transition increases the number and adjusts e accordingly by switching e from true to false and vice versa. The second transition halts the machine by setting $n = 0$ such that no further transitions are possible.

Formally the finite state machine $\widetilde{M}$ can be written as $\widetilde{M} = (\{\hat{m}\}, \hat{m}, V, \hat{v}, \rightarrow)$ with

- $V = \{n : \mathbb{N}_0, e : \{true, false\}\}$,
- $\hat{v} = (n = 1) \wedge (e = false)$, and
- $\rightarrow = \{(\hat{m}, [n > 0], n \mapsto n + 1; e \mapsto \neg e, \hat{m}), (\hat{m}, [n > 0], n \mapsto 0, \hat{m})\}$.

Example 2.1.2. The formula to calculate any weakest pre-condition of a variable assignment v in the finite state machine $\widetilde{M}$ is:

- $\text{pre}(v) = ((n > 0) \wedge v[n \mapsto 0]) \vee ((n > 0) \wedge v[n \mapsto n + 1; e \mapsto \neg e])$.

More examples of the actual calculation of the weakest pre-condition can be seen at the end of Example 2.2.1 on page 7.

2.2 Pointed Kripke Structures

In accordance to Fecher and Shoham [7] I will not consider action labels on models, thus the concrete models considered here are pointed Kripke structures over a predicate language $\mathcal{L}$. Such structures are directed graphs. Arcs represent *transitions* and vertices represent *worlds*. Propositions taken from the boolean closure of a predicate language $\mathcal{L}$ are assigned as labels to each vertex.

With regard to finite state machines, each world of a pointed Kripke structure represents one possible assignment of variables. Each transition represents a possible change in this assignment.

Definition 2.2.1 (Pointed Kripke structure)**.** A *pointed Kripke structure* K is a tuple $K = (W, \hat{w}, R, \mathcal{L})$ where

- W is a (nonempty) set of worlds,
- $\hat{w} \in W$ is an initial world,
- $R \subseteq W \times W$ is its world transition relation, and
- $\mathcal{L}$ is a *good predicate language* as defined in Definition 2.2.4 on the next page.

K is finite whenever W is.

Definition 2.2.2 (Predicate language $\mathcal{L}$)**.** A *predicate language* $\mathcal{L}$ is a set of predicates interpreted over the worlds W:

- $\mathcal{L}$ is a (nonempty) set of predicates p, and
- there exists a function $\langle\!|\cdot|\!\rangle$ with $\langle\!|\cdot|\!\rangle : \mathcal{L} \to \mathbb{P}(W)$.

Thus each predicate $p \in \mathcal{L}$ denotes a set $\langle\!|p|\!\rangle \subseteq W$ of worlds. Usually $\langle\!|p|\!\rangle$ is the set of worlds where the predicate p holds true.

Let $\overline{\mathcal{L}}$ be the boolean closure of the predicate language $\mathcal{L}$.

Definition 2.2.3 (Extension function $[\![\cdot]\!]$)**.** The extension of the function $\langle\!|\cdot|\!\rangle$ to $\overline{\mathcal{L}}$ is called the extension function $[\![\cdot]\!]$. Thus the *extension function* $[\![\cdot]\!] : \overline{\mathcal{L}} \to \mathbb{P}(W)$ for a pointed Kripke structure $(W, \hat{w}, R, \mathcal{L})$ with $\psi, \psi' \in \overline{\mathcal{L}}$ (where $\overline{\mathcal{L}}$ is the boolean closure of $\mathcal{L}$) is defined as:

- for all $p \in \mathcal{L}$: $[\![p]\!] = \langle\!|p|\!\rangle$,
- $[\![false]\!] = \emptyset$ and $[\![true]\!] = W$,
- $[\![\psi \vee \psi']\!] = [\![\psi]\!] \cup [\![\psi']\!]$,
- $[\![\psi \wedge \psi']\!] = [\![\psi]\!] \cap [\![\psi']\!]$, and
- for all $w \in W$: $w \notin [\![\psi]\!]$ iff $w \in [\![\neg\psi]\!]$.

Thus for every predicate $p \in \mathcal{L}$ and every world $w \in W$ one has either $w \in [\![p]\!]$ or $w \in [\![\neg p]\!]$.

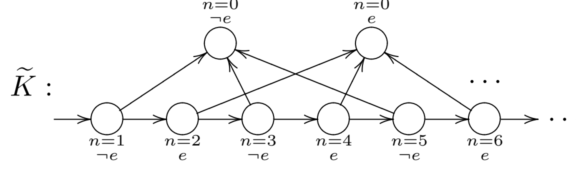

Figure 2.2: A *pointed Kripke structure*. Circles depict worlds $w \in W$, arrows $w \to w'$ denote transitions $(w, w') \in R$. The predicates $p \in \overline{\mathcal{L}}$ are depicted in small font near the corresponding worlds while world names are omitted for the sake of simplicity.

Definition 2.2.4 (Good predicate language $\mathcal{L}$)**.** A predicate language $\mathcal{L}$ is called *good* if there exists an extension function $[\![\cdot]\!]$ and

- there exists $\hat{p} \in \overline{\mathcal{L}}$ with $[\![\hat{p}]\!] = \{\hat{w}\}$,
- the boolean closure $\overline{\mathcal{L}}$ of $\mathcal{L}$ is a decidability theory (i.e. satisfiability is decidable),
- $\overline{\mathcal{L}}$ is closed under exact predecessor operations in pointed Kripke structures, i.e., for every formula $\psi \in \overline{\mathcal{L}}$ one can compute the weakest precondition $\text{pre}(\psi) \in \overline{\mathcal{L}}$ such that

$$[\![\text{pre}(\psi)]\!] = \{w \in W \mid \exists w' \in [\![\psi]\!] : (w, w') \in R\}.$$

One can utilize the related finite state machine to calculate the weakest precondition efficiently.

Example 2.2.1. Consider the simple program from Example 2.1.1 on page 5 to determine if a given number ($\in \mathbb{N}$) is even.

A possible pointed Kripke structure $\widetilde{K}$ to model such a system is illustrated in Figure 2.2. Here the set of worlds W consists of the set of worlds $\tilde{w}_i$ for $i \in \mathbb{N}$ and the two worlds $\tilde{w}'$ and $\tilde{w}''$ which represent printing the result for even or odd numbers. The initial world is $\tilde{w}_1$. There are two sets of predicates used to characterize these worlds: $n = 0, 1, 2, \dots$ represents the counting variable with $[\![n = 0]\!] = \{\tilde{w}', \tilde{w}''\}$ and $[\![n = i]\!] = \{\tilde{w}_i\}$ for all $i \in \mathbb{N}$. The second set of predicates, $e = \mathit{true}$ and $e = \mathit{false}$ (or e and $\neg e$), means even and not even respectively, and is assigned accordingly, thus $[\![e]\!] = \{\tilde{w}''\} \cup \{\tilde{w}_{2i} | i \in \mathbb{N}\}$ and $[\![\neg e]\!] = \{\tilde{w}'\} \cup \{\tilde{w}_{2i-1} | i \in \mathbb{N}\}$. The set of transitions R is depicted in the figure.

To check a given number m one starts at $\tilde{w}_1$, and if $m \neq 1$ takes the transition from $\tilde{w}_1$to $\tilde{w}_2$ and continues onward from $\tilde{w}_{n-1}$to $\tilde{w}_n$ until $n = m$ holds true, thus reaching $\tilde{w}_m$. Then one finally takes the transition to $\tilde{w}'$ or $\tilde{w}''$ respectively and gets the result from the value of e.

Formally the pointed Kripke structure $\widetilde{K}$ is defined as $\widetilde{K} = (W, \tilde{w}_1, R, \mathcal{L})$ with

- $W = \{\tilde{w}', \tilde{w}''\} \cup \{\tilde{w}_i \mid i \in \mathbb{N}\}$,
- $R = \{(\tilde{w}_{2i}, \tilde{w}'') \mid i \in \mathbb{N}\} \cup \{(\tilde{w}_{2i-1}, \tilde{w}') \mid i \in \mathbb{N}\} \cup \{(\tilde{w}_i, \tilde{w}_{i+1}) \mid i \in \mathbb{N}\}$,
- $\mathcal{L}$ is a good predicate language as described below.

The good predicate language $\mathcal{L}$ is formally defined as

- $\mathcal{L} = \{e, n > 0, n = 0, n = 1, n = 2, \ldots\}$ with
  - $\langle\!\langle e \rangle\!\rangle = \{\tilde{w}''\} \cup \{\tilde{w}_{2i} \mid i \in \mathbb{N}\}$,
  - $\langle\!\langle n > 0 \rangle\!\rangle = \{\tilde{w}_i \mid i \in \mathbb{N}\}$, and
  - $\langle\!\langle n = i \rangle\!\rangle = \{\tilde{w}_i\}$ for all $i \in \mathbb{N}$.

Thus the boolean closure $\overline{\mathcal{L}}$ contains predicates like

- $\neg e$ or $(n > 0) \wedge \neg e$ with
- $[\![\neg e]\!] = \{\tilde{w}'\} \cup \{\tilde{w}_{2i-1} \mid i \in \mathbb{N}\}$,
- $[\![n > 0 \wedge \neg e]\!] = \{\tilde{w}_{2i-1} \mid i \in \mathbb{N}\}$.

Finally $\mathcal{L}$ is a *good* predicate language, since

- there exists $(n = 1) \wedge e \in \overline{\mathcal{L}}$ with $[\![(n = 1) \wedge e]\!] = \{\tilde{w}_1\} = \{\hat{w}\}$,
- Satisfiability of terms such as
  - $SAT(n > 0) = true$ or $SAT((n > 0) \wedge (n = 0)) = false$

  is easily calculated.
- CPL is closed under the exact predecessor operation, e.g.
  - $\text{pre}(n = 0) = (n > 0)$,
  - $\text{pre}(n > 0) = (n > 0)$,
  - $\text{pre}(e) = (n > 0)$,
  - $\text{pre}(\neg e) = (n > 0)$,
  - $\text{pre}(e \wedge n = 0) = e \wedge (n > 0)$,
  - $\text{pre}(e \wedge n > 0) = \neg e \wedge (n > 0)$,
  - et cetera

The calculation of these pre-conditions can efficiently be performed with the help of the finite state machine $\widetilde{M}$ from Example 2.1.1 and 2.1.2 on page 5. A few examples:

$$\text{pre}(\phi) = ((n > 0) \wedge \phi[n \mapsto 0]) \vee ((n > 0) \wedge \phi[n \mapsto n + 1; e \mapsto \neg e])$$

$$\begin{aligned}
\text{pre}(n = 0) &= ((n > 0) \wedge (n = 0)[n \mapsto 0]) \vee ((n > 0) \wedge (n = 0)[n \mapsto n + 1; e \mapsto \neg e]) \\
&= ((n > 0) \wedge (0 = 0)) \vee ((n > 0) \wedge (n + 1 = 0)) \\
&= (n > 0) \vee ((n > 0) \wedge (n = -1)) \\
&= (n > 0)
\end{aligned}$$

$$\begin{aligned}
\text{pre}(true) &= ((n > 0) \wedge true[n \mapsto 0]) \vee ((n > 0) \wedge true[n \mapsto n + 1; e \mapsto \neg e]) \\
&= ((n > 0) \wedge true) \vee ((n > 0) \wedge true) \\
&= (n > 0)
\end{aligned}$$

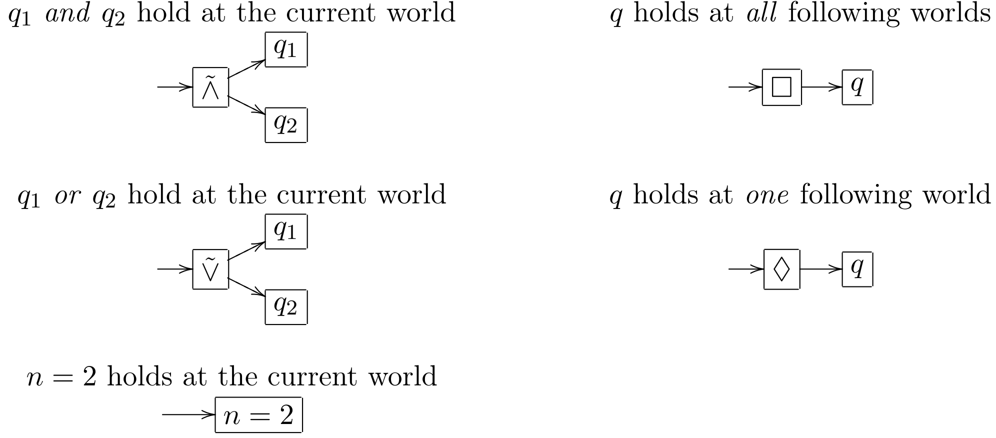

Figure 2.3: Transitions in *alternating tree automata*. $q, q_1, q_2 \in Q$, $(n = 2) \in \mathcal{L}$.

2.3 Alternating Tree Automata

The *modal μ-calculus* [15] is a logic to express properties of pointed Kripke structures. Here I use the modal μ-calculus in its equivalent form of alternating tree automata. This equivalence is shown by Wilke [19].

Definition 2.3.1 (Alternating tree automaton)**.** An *alternating tree automaton* for a property language $\mathcal{L}$ is a tuple $A = (Q, \hat{q}, \delta, \theta)$ such that

- Q is a finite, nonempty set of states,
- $\hat{q} \in Q$ is an initial state,
- δ is a transition function, which maps each automaton state ($\in Q$) to one of the following forms: $p \mid q' \mid q'\tilde{\vee}q'' \mid q'\tilde{\wedge}q'' \mid \Diamond q' \mid \Box q'$ (where $p \in \mathcal{L}; q', q'' \in Q$), and
- $\theta \colon Q \to \mathbb{N}$ is an acceptance condition, which assigns an acceptance number to each state.

Figure 2.3 illustrates the meaning of an automaton state in the context of a world of a pointed Kripke structure. The next definition introduces some useful notations (for subsets of the states of an alternating tree automaton) that will become meaningful in connection with property games.

Definition 2.3.2 (Useful notations)**.** $Q_0, Q_1, Q_{0/1} \subseteq Q$ are defined as

- $Q_0 = \{q \in Q \mid \delta(q) \in \bigcup_{q',q'' \in Q}\{q', q'\tilde{\vee}q'', \Diamond q'\}\}$,
- $Q_1 = \{q \in Q \mid \delta(q) \in \bigcup_{q',q'' \in Q}\{q'\tilde{\wedge}q'', \Box q'\}\}$,
- $Q_{0/1} = \{q \in Q \mid \delta(q) \in \mathcal{L}\}$,
- $Q_{quan} = \{q \in Q \mid \delta(q) \in \bigcup_{q' \in Q}\{\Diamond q', \Box q'\}\}$, and
- $Q_{\overline{quan}} = \{q \in Q \mid \delta(q) \in \bigcup_{p \in \mathcal{L}, q',q'' \in Q}\{p, q', q'\tilde{\vee}q'', q'\tilde{\wedge}q''\}\}$.

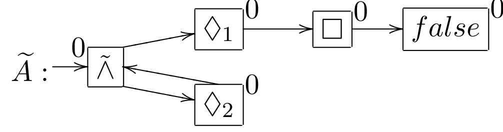

Figure 2.4: An *alternating tree automaton.* States are depicted as rectangles. A depiction of the transition function is within the state borders in connection with the transition arrows. Acceptance numbers are depicted close to corresponding states, state names are omitted for the sake of simplicity. To differentiate between both $\Diamond$-states a subscript is added.

Additionally, combinations of these notations are used, such as $Q_0 \cap Q_{quan}$.

Example 2.3.1. Considering the finite state machine in Example 2.1.1 on page 5 and the associated pointed Kripke structure given in Example 2.2.1 on page 7 one might be interested whether the given system is able to reach numbers ($\in \mathbb{N}$) as well as to end execution at each of them. So, at each world $\tilde{w}_i$ there must be a transition leading to one of the two final worlds and one transition to next world $\tilde{w}_{i+1}$.

In other words: $(*)$ (i) there is a transition such that no further transition is possible; and (ii) there is a transition such that $(*)$ holds again. Note that these transitions are inside the pointed Kripke structure.

An example of an alternating tree automaton to express $(*)$ is depicted in Figure 2.4. At the current world, there are two ($\tilde{\wedge}$) transitions ($\Diamond_1$ and $\Diamond_2$): One transition in which the starting property holds again ($\Diamond_2$) and one transition in which no further transition exists ($\Diamond_1$). More precisely for all further transitions ($\Box$) $false$ holds true, which it never does, thus there can't be any further transitions.

Note that the term transitions refers to the pointed Kripke structure. Every time the transition function of the alternating tree automata refers to $\Diamond$ or $\Box$, a transition in the pointed Kripke structure is taken. This will be clarified further by Section 2.4 Parity Games on the next page.

Formally the alternating tree automata $\widetilde{A}$ is defined as $\widetilde{A} = (Q, \tilde{q}, \delta, \theta)$ with

- $Q = \{\tilde{q}, \tilde{q}_1, \tilde{q}_1', \tilde{q}_1'', \tilde{q}_2\}$,
- $\delta\colon \tilde{q} \mapsto \tilde{q}_1 \tilde{\wedge} \tilde{q}_2; \tilde{q}_1 \mapsto \Diamond\, \tilde{q}_1'; \tilde{q}_1' \mapsto \Box\, \tilde{q}_1''; \tilde{q}_1'' \mapsto false; \tilde{q}_2 \mapsto \Diamond\, \tilde{q}$,
- $\theta\colon Q \to \mathbb{N}$ with $q \mapsto 0$ for all $q \in Q$.

Furthermore Q_0, Q_1 and $Q_{0/1}$ are defined as follows:

$$Q_0 = \{\tilde{q}_1, \tilde{q}_2\},\ Q_1 = \{\tilde{q}, \tilde{q}_1'\},\ \text{and } Q_{0/1} = \{\tilde{q}_1''\}.$$

2.4 Parity Games

Parity games are special kinds of (infinite) games, played by two players (*Player 0* and *Player 1*) on a graph. Every game position is represented by vertices, possible moves are represented by arcs. Additionally every game position is assigned a priority (i.e. a natural number).

The set of vertices C is partitioned into three disjoint subsets C_0 and C_1 representing game positions belonging to Player 0 and Player 1, respectively, and $C_{0/1}$ representing additional game positions, usually not belonging to any player.

Each turn one player does a move according to a fixed set of rules. Usually the player in possession of the current game position selects an outgoing arc leading to a new game position.

A finite play of a parity game is usually won by Player 0 if Player 1 is unable to move and vice versa. Infinite plays are won by the Player 0 if the highest infinitely often occurring priority is even, otherwise Player 1 wins. Additional winning or losing conditions can be specified by the rules.

2.4.1 Strong-Weak-Parity-Games

In accordance with Fecher and Shoham [7] a generalized form of three-valued parity games (Grumberg et al. [12]) is employed.

Definition 2.4.1 (Strong-weak-parity-game)**.** A *strong-weak-parity-game* is defined as a tuple $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ with

- $C_0 \subseteq C$ is the set of game position belonging to Player 0,
- $C_1 \subseteq C$ is the set of game position belonging to Player 1,
- $C_{0/1} \subseteq C$ is the set of game position belonging neither to Player 0 nor to Player 1,
- $\hat{c} \in C$ is the initial game position,
- $G^-, G^+, G^\circ \subseteq C \times C$ is the set of strong, weak and junction game transitions, respectively,
- $\vartheta \colon C \to \mathbb{N}$ is a parity function with finite image, and
- $\omega \colon C \to \{tt, ff, \bot\}$ is a validity function into the values *true*, *false*, and *unknown*.

As above $C = C_0 \,\dot{\cup}\, C_1 \,\dot{\cup}\, C_{0/1}$ is the set of all game positions.

Definition 2.4.2 (Validity plays)**.** The rules for *validity plays* of strong-weak-parity-games are presented below. If the current game position is c, choose the rule detailing winning conditions or movement according to the value of $\omega(c)$ and according to the subset c belongs to:

$$\begin{aligned}
\omega(c) \neq \bot &\colon \text{ Player 0 wins iff } \omega(c) = tt\text{; otherwise Player 1 wins;}\\
c \in C_0 \wedge \omega(c) = \bot &\colon \text{ Player 0 picks as next game position } c' \in \{c\}.(G^- \cup G^\circ);\\
c \in (C_{0/1} \cup C_1) \wedge \omega(c) = \bot &\colon \text{ Player 1 picks as next game position } c' \in \{c\}.(G^+ \cup G^\circ);
\end{aligned}$$

In addition to the winning conditions stated by the rules, a *finite* validity play is won by a Player 0, if Player 1 can not move at a game position $c \in C_1$. Otherwise Player 1 wins. An *infinite* validity play is won by Player 0 iff the maximum of all infinitely often occurring parity numbers ($\vartheta(c) \in \mathbb{N}$) is even, otherwise it is won by Player 1.

Definition 2.4.3 (Invalidity plays)**.** The rules for *invalidity plays* of strong-weak-parity-games are presented below. If the current game position is c, choose the rule detailing winning conditions or movement according to the value of $\omega(c)$ and according to the subset c belongs to:

$$\begin{aligned}
\omega(c) \neq \bot &: \text{ Player 1 wins iff } \omega(c) = \mathit{ff}\text{; otherwise Player 0 wins;} \\
c \in C_1 \wedge \omega(c) = \bot &: \text{ Player 1 picks as next game position } c' \in \{c\}.(G^- \cup G^\circ); \\
c \in (C_{0/1} \cup C_0) \wedge \omega(c) = \bot &: \text{ Player 0 picks as next game position } c' \in \{c\}.(G^+ \cup G^\circ);
\end{aligned}$$

In addition to the winning conditions stated by the rules, a *finite* invalidity play is won by a Player 1, if Player 0 is unable to move at a game position $c \in C_0$. Otherwise Player 0 wins. An *infinite* invalidity play is won by Player 1 iff the maximum of all infinitely often occurring parity numbers ($\vartheta(c) \in \mathbb{N}$) is odd, otherwise it is won by Player 0.

Definition 2.4.4 (Winning strategy)**.** A player is said to have a *winning strategy* iff he has a strategy to choose the movement that allows him to win every play regardless of moves the other player performs.

Definition 2.4.5 (Valid and invalid games)**.**

- The strong-weak-parity-game G is *valid* (*invalid*) *in* $c \in C$ iff Player 0 (Player 1) has a winning strategy for the corresponding validity (invalidity) play starting in c.
- The strong-weak-parity-game G is *valid* (*invalid*) iff G is valid (invalid) in $\hat{c}$.

Three-valued parity games are a superset of normal (two-valued) parity games. Two-valued parity games are either valid or invalid, i.e if Player 0 has no winning strategy to win all validity plays then Player 1 has a winning strategy to win all invalidity plays and vice versa. Due to abstraction (as introduced in the next chapter) this is no longer sufficient. A game may be not-valid *and* not-invalid, thus unknown, i.e. the system is too much abstracted to verify or to refute the property. As mentioned in the introduction this will lead to refinement.

Definition 2.4.6. A strong-weak-parity-game G is called *well-formed* iff $(c_1, c_2) \in G^-$ implies $(c_1, c_2) \in G^+$, thus $G^- \subseteq G^+$.

In the following I will only consider well-formed strong-weak-parity-games.

Theorem 2.4.1. *Validation over well-formed strong-weak-parity-games is three-valued, thus a strong-weak-parity-game is either valid, invalid or neither of them.*

Proof. A simple strong-weak-parity-game that is neither valid nor invalid is the game $G_x = (\{x\}, \emptyset, \emptyset, x, \emptyset, \{(x,x)\}, \emptyset, x \mapsto 0, x \mapsto \bot)$. In this game there exists a single C_0 game position x with the parity number 0 and unknown validity. The sole game transition is a weak game transition from x to x. Player 0 looses all validity plays since he cannot move. Furthermore Player 1 looses all invalidity plays since Player 0 can always move, leading to an infinite game, which is won by Player 0. Thus G_x is neither valid nor invalid.

The next things to prove is: if a game is valid it cannot be invalid and if a game is invalid it cannot be valid.

Let G be a valid game. So Player 0 has a winning strategy for the validity game. With the help of this strategy he wins every validity play regardless of the moves Player 1 performs at game positions in C_1 and in $C_{0/1}$. In the validity play Player 0 only picks if the current game position is in C_0 and the only game transitions he may utilize are strong and junction game transitions.

In the invalidity play Player 0 may pick at game positions in $C_0 \cup C_{0/1}$. Furthermore, instead of strong game transitions he must now use weak game transitions. Regardless of that fact, Player 0 can still apply his winning strategy. The strategy is independent of moves performed by Player 1, thus at a game position in $C_{0/1}$ Player 0 just acts randomly (Player 1 would pick here in the validity game). At game positions in C_0 Player 0 just acts according to the winning strategy, with one exception: If Player 0 would pick a strong game position he instead picks a weak game position. This is feasible since G is well-formed. Thus Player 0 wins all invalidity plays and the game is not invalid.

A similar line of reason holds for G as an invalid game. □

2.4.2 Property Games

Satisfaction of a pointed Kripke structure with respect to a property language (here the modal μ-calculus in its equivalent form of *alternating tree automata*) is obtained with the help of a special strong-weak-parity-game called *property game*.

Definition 2.4.7 (Property game)**.** A *property game* for a pointed Kripke structure $K = (W, \hat{w}, R, \mathcal{L})$ and an alternating tree automaton $A = (Q, \hat{q}, \delta, \theta)$ is defined as the strong-weak-parity-game $G_{K,A} = (W \times Q_0, W \times Q_1, W \times Q_{0/1}, (\hat{w}, \hat{q}), G^-, G^+, G^\circ, \vartheta, \omega)$, where

- $G^- = G^+ = \{((w,q),(w',q')) \mid \delta(q) \in \{\Diamond q', \Box q'\} \wedge (w,w') \in R\}$,
- $G^\circ = \{((w,q),(w,q')) \mid \exists q'' : \delta(q) \in \{q', q'\tilde{\vee} q'', q''\tilde{\vee} q', q'\tilde{\wedge} q'', q''\tilde{\wedge} q'\}\}$,
- $\vartheta(w,q) = \theta(q)$, and
- $\omega(w,q) = \begin{cases} tt, & \text{if } q \in Q_{0/1} \wedge w \in [\![\delta(q)]\!] \\ ff, & \text{if } q \in Q_{0/1} \wedge w \notin [\![\delta(q)]\!] \\ \bot, & \text{otherwise.} \end{cases}$

Definition 2.4.8 (Satisfaction for pointed Kripke structures)**.** A pointed Kripke structure K *satisfies* an alternating tree automaton A, written $K \models A$ iff the corresponding property game $G_{K,A}$ is valid.

Since $G^- = G^+$ one can easily see that $G_{K,A}$ is valid iff it is not invalid, and vice versa. Furthermore, this definition of satisfaction for pointed Kripke structures corresponds to the one presented by Wilke [19] in Sec. 3 of his paper. Equivalence of the definition and the rules to the well known satisfaction of Kripke Structures with the modal μ-calculus follows from Theorem 3 in the same paper.

Example 2.4.1. Considering the pointed Kripke structure $\widetilde{K}$ in Example 2.2.1 on page 7 and the alternating tree automaton $\widetilde{A}$ in Example 2.3.1 on page 10, one can construct the associated property game $G_{\widetilde{K},\widetilde{A}}$.

A section of this property game is illustrated in Figure 2.5 on the following page. Since $\widetilde{K}$ is not finite, the resulting property game $G_{\widetilde{K},\widetilde{A}}$ cannot be finite, too. This is illustrated by $\cdots$. Put simply, the section of the game repeats its self ad infinitum. Additionally, unreachable game positions are not depicted, such as $(\tilde{w}_1, \tilde{q}''_1) \in W \times Q_{0/1}$ or as it is depicted in the figure $\binom{n=1\wedge\neg e}{false}$. For the rest of this example I will use the less formal notion used in the figure to denote game positions.

Nevertheless this section of the property game can be used to sketch a winning strategy for Player 0:

- $\binom{n=i\wedge\neg e}{\Diamond_1}$: Player 0 picks $\binom{n=0\wedge\neg e}{\Box}$,
- $\binom{n=i\wedge e}{\Diamond_1}$: Player 0 picks $\binom{n=0\wedge e}{\Box}$,
- $\binom{n=i\wedge\neg e}{\Diamond_2}$: Player 0 picks $\binom{n=i+1\wedge e}{\tilde{\wedge}}$, and
- $\binom{n=i\wedge e}{\Diamond_2}$: Player 0 picks $\binom{n=i+1\wedge\neg e}{\tilde{\wedge}}$.

With this strategy, all finite plays end as soon as a game position $\binom{n=i\wedge\neg e}{\Diamond_1}$ or $\binom{n=i\wedge e}{\Diamond_1}$ is reached. Player 0 picks $\binom{n=0\wedge\neg e}{\Box}$ or $\binom{n=0\wedge e}{\Box}$, Player 1 will be forced to move, but is not able to do so. Also, all plays that never reach one of the game positions mentioned above are infinite. These plays are also won by Player 0 since the maximum over all infinitely often occurring parity numbers is zero, which is even. Thus $\widetilde{K}$ satisfies $\widetilde{A}$.

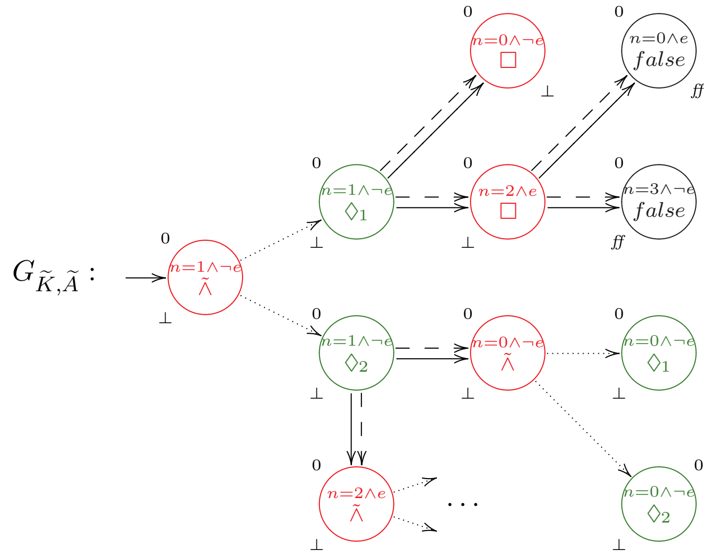

Figure 2.5: A *property game*. Green circles are C_0 game positions, red circles are C_1 game positions, and black circles are $C_{0/1}$ game positions. Values of the parity function ϑ and values of the validity function ω are depicted near the corresponding game positions. Solid arrows depict strong game transitions, dashed arrows weak game transitions, and dotted arrows junction game transitions. World names as well as state names are omitted, instead the predicates ($\in \overline{\mathcal{L}}$) and transition symbols ($\in \delta$) are depicted.

Chapter 3

Underlying Structures and Methods

3.1 Predicate Abstraction

Predicate abstraction is a technique used to counter the large (possibly infinite) number of game positions inherent to the construction of property games with pointed Kripke structures. Here, predicate abstraction is used to reduce the number of worlds in a given pointed Kripke structure, thus reducing this concrete structure into an abstract one.

Definition 3.1.1 (Abstraction function)**.** Let K be a pointed Kripke structure, with $K = (W, \hat{w}, R, \mathcal{L})$ and let I be a set of abstract worlds usually with $|I| \leq |W|$. An *abstraction function* α is a total function $\alpha : I \to \overline{\mathcal{L}}$ which assigns a formula of $\overline{\mathcal{L}}$ to each abstract world.

With the help of the extension function $[\![\cdot]\!]$ (see Definition 2.2.3 on page 6) one can determine the set of concrete worlds ($\subset W$) which are *abstracted* into an abstract world $i \in I$. Thus all $w \in [\![\alpha(i)]\!]$ for an $i \in I$ are abstracted into the same abstract world i. Similarly two worlds $w, w' \in W$ are said to be *compatible* iff there exists an $i \in I$ with $w \in [\![\alpha(i)]\!]$ and $w' \in [\![\alpha(i)]\!]$.

Example 3.1.1. Let $\tilde{\alpha} : \tilde{I} \to \overline{\mathcal{L}}$ be an abstraction function for the pointed Kripke structure $\widetilde{K}$ described in Example 2.2.1 on page 7 with $\tilde{I} = \{\tilde{i}, \tilde{i}', \tilde{i}''\}$ and

- $\tilde{\alpha}(\tilde{i}) = (n > 0)$,
- $\tilde{\alpha}(\tilde{i}') = (n = 0) \wedge \neg e$, and
- $\tilde{\alpha}(\tilde{i}'') = (n = 0) \wedge e$.

Thus, with the help of the extension function $[\![\cdot]\!]$, one gains the concrete worlds ($\in W$), abstracted by the elements of $\tilde{I}$:

$$\begin{aligned}
\tilde{i}: \quad & [\![\tilde{\alpha}(\tilde{i})]\!] = [\![n > 0]\!] && = \{\tilde{w}_n \mid n \in \mathbb{N}\}, \\
\tilde{i}': \quad & [\![\tilde{\alpha}(\tilde{i}')]\!] = [\![(n = 0) \wedge \neg e]\!] && = \{\tilde{w}'\}, \\
\tilde{i}'': \quad & [\![\tilde{\alpha}(\tilde{i}'')]\!] = [\![(n = 0) \wedge e]\!] && = \{\tilde{w}''\},
\end{aligned}$$

Clearly all $\tilde{w}_n, n \in \mathbb{N}$ are compatible worlds under $\tilde{\alpha}$.

3.2 Abstraction Techniques

Let K_I^α be an *abstracted structure*. This means K_I^α is derived from a pointed Kripke structure K by applying the abstraction function α to I and adjusting the rest of the structure in a defined way. Fecher and Huth [6] utilized an abstraction technique called pre-abstraction, which uses generalized Kripke modal transition systems as abstracted structures, as well as a technique called post-abstraction, using μ-automata as abstracted structures. I developed a combination of these two abstraction techniques utilizing a new kind of automaton, called *generalized μ-automaton* [18], as the abstracted structure.

An abstraction technique is called *sound* iff for all alternating tree automata A, the abstracted structure satisfies A does imply that the concrete pointed Kripke structure satisfies A. Generally, the opposite implication does not hold.

In the following, let $K = (W, \hat{w}, R, \mathcal{L})$ be a pointed Kripke structure, I a set of abstract worlds, $\alpha : I \to \overline{\mathcal{L}}$ an abstraction function, and $A = (Q, \hat{q}, \delta, \theta)$ an alternating tree automaton.

3.2.1 Pre-Games and Pre-Abstraction

Pre-games are similar to property-games as defined for pointed Kripke structures (see Definition 2.4.7 on page 13). Each game position is a pair (i, q), where $i \in I$ is an abstract world and $q \in Q$ is an alternating tree automaton state. Before applying any rule (so the name *pre*) of a validity or invalidity play, Player 1 (or Player 0) chooses one concrete world w with $w \in [\![\alpha(i)]\!]$. This world is used to determine the rule in the property game $G_{K,A}$ at the game position (w, q), resulting in (w', q'). Then the new game position in the pre-game is (i', q') with $w' \in [\![\alpha(i')]\!]$.

Hence one can only validate or invalidate properties encoded in A that hold for all compatible worlds.

Thus the idea is to base the game upon an abstract structure, while performing all actual moves upon the concrete structure. This leads to *pre-abstraction*, which uses *generalized Kripke modal transition systems* incorporating *must-hypertransitions* to implement the abstraction given by α as well as the possible world changes inherent to pre-games.

Definition 3.2.1 (Generalized Kripke modal transition systems)**.** A *generalized Kripke modal transition systems* G over $\mathcal{L}$ is a tuple $(T, \hat{t}, R^-, R^+, L)$ where

- T is a (nonempty) set of worlds,
- $\hat{t} \in T$ is an initial world,
- $R^- \subseteq T \times \mathbb{P}(T)$ is its set of must-hypertransitions,
- $R^+ \subseteq T \times T$ is its set of may-transitions, and
- $L\colon T \to \overline{\mathcal{L}}$ is its labeling function.

G is finite whenever T is.

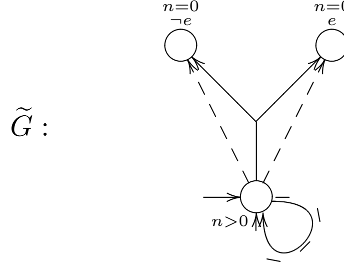

Figure 3.1: A *generalized Kripke modal transition system.* Circles depict worlds $t \in T$, dashed arrows $t \dashrightarrow t'$ denote may transitions $(t, t') \in R^+$, while solid arrows $t \to \{t_1, \ldots, t_n\}, n \in \mathbb{N}$ represent must-hypertransitions $(t, \{t_1, \ldots, t_n\}) \in R^-$. Labels $L(t) \in \overline{\mathcal{L}}$ are depicted in small font near the corresponding worlds, while world names are omitted for the sake of simplicity.

The complete means of constructing the generalized Kripke modal transition system via *pre-abstraction* from a pointed Kripke structure and an abstraction function are left to the paper of Fecher and Huth [6], Definition 7.

Soundness of pre-abstraction is shown by Fecher and Huth [6].

Example 3.2.1. An example of a generalized Kripke modal transitions system resulting from the pointed Kripke structure in Example 2.2.1 on page 7 and the abstraction function $\tilde{\alpha}$ and the set of abstract worlds $\tilde{I}$ from Example 3.1.1 on page 16 can be seen in Figure 3.1.

As one would expect from the abstraction function $\tilde{\alpha}$, all concrete worlds w_n for $n \in \mathbb{N}$ are abstracted into a single abstract world $\tilde{i}$. The labeling function simply corresponds to the abstraction function.

There exists a may-transition $(i, i') \in R^+$ if there exists at least one transition from any concrete world abstracted by i to any concrete worlds abstracted by i'. There exists a must-hypertransition $(i, \{i_1, \ldots, i_n\}) \in R^-$ if there exists a transition from all concrete worlds abstracted by i to a concrete world abstracted by one abstract world from $\{i_1, \ldots, i_n\}$.

Formally the generalized Kripke modal transitions system $\widetilde{G}$ is defined as:
$\widetilde{G} = (T, \tilde{i}, R^-, R^+, L)$ with

- $T = \{\tilde{i}, \tilde{i}', \tilde{i}''\} = \tilde{I}$,
- $R^- = \{(\tilde{i}, \{\tilde{i}\}), (\tilde{i}, \{\tilde{i}', \tilde{i}''\})\}$,
- $R^+ = \{(\tilde{i}, \tilde{i}), (\tilde{i}, \tilde{i}'), (\tilde{i}, \tilde{i}'')\}$, and
- $L(t) = \tilde{\alpha}(t)$ for all $t \in T$.

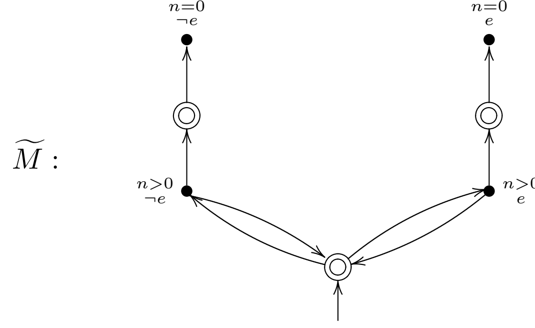

Figure 3.2: A *μ-automaton*. OR-worlds ($\in O$) are depicted as double-lined circles and BRANCH-worlds ($\in B$) as small solid circles. $L(b) \in \overline{\mathcal{L}}$ at a BRANCH-world b are depicted in small font next to the corresponding worlds. Arrows $o \rightarrow b$ and $b \rightarrow o$ denote OR-relations $(o, b) \in \rightrightarrows$ respectively BRANCH-relations $(b, o) \in \rightarrow$. World names are omitted for the sake of simplicity.

3.2.2 Post-Games and Post-Abstraction

The idea of post-games is similar to the idea of pre-games. Instead of switching to a compatible world prior to a move in the game, one switches after every move of the game (hence the name *post*). Each game position is a pair (w, q) with $w \in W$ and $q \in Q$. After applying any rule of a validity or invalidity play in the property game $G_{K,A}$, resulting in game position (w', q'), Player 1 (or Player 0) chooses one compatible world $w'' \in W$ (i.e. $\exists i \in I : w', w'' \in [\![\alpha(i)]\!]$). Then the new game position in the pre-game is (w'', q'). Hence one is usually able to validate or invalidate more properties than in pre-games, albeit at a usually higher cost.

This leads to *post-abstraction*, which uses *μ-automata* to implement the abstraction given by α as well as the possible world changes inherent to post-games.

Definition 3.2.2 (μ-automata)**.** A *μ-automaton* M over $\mathcal{L}$ is a tuple $(O, B, \hat{w}, \rightrightarrows, \rightarrow, L)$ such that

- O is a set of OR-worlds,
- B is a set of BRANCH-worlds (disjoint from O),
- $\hat{w} \in O \cup B$ is an initial world,
- $\rightrightarrows \subseteq O \times B$ is its OR-transition relation,
- $\rightarrow \subseteq B \times O$ is its BRANCH-transition relation, and
- $L \colon B \rightarrow \overline{\mathcal{L}}$ is its labeling function.

M is finite if both B and O are.

The complete means of constructing μ-automata via *post-abstraction* from a pointed Kripke structure and an abstraction function are left to the paper of Fecher and Huth [6], Definition 10.

Soundness of post-abstraction is shown by Fecher and Huth [6].

Example 3.2.2. An example of a μ-automaton resulting from the pointed Kripke structure in Example 2.2.1 on page 7 and the abstraction function $\tilde{\alpha}$ and the set of abstract worlds $\tilde{I}$ from Example 3.1.1 on page 16 can be seen in Figure 3.2 on the previous page.

Similar to Example 3.2.1 on page 18 all concrete worlds w_n for $n \in \mathbb{N}$ are abstracted into a single abstract OR-world $\tilde{\imath}$. Seen altogether, the set of OR-worlds corresponds to the set of abstract worlds $\tilde{I}$. Additionally, there is a set of BRANCH-worlds for each OR-world. Each element of such a set corresponds to an equivalence class of concrete worlds abstracted by the particular OR-world. Each BRANCH-world (equivalence class) is a tuple consisting of two parts. As mentioned before, each BRANCH-world is assigned to a subset of the concrete worlds abstracted by the OR-world. The first part of the BRANCH-tuple is the set of abstract OR-worlds reachable from the respective BRANCH-world (the abstract worlds that abstract the concrete worlds that are reachable by the concrete worlds assigned to the BRANCH-world). The second part is the formula ($\in \overline{\mathcal{L}}$) that holds true at all concrete worlds belonging to the equivalence class. For a more detailed explanation of the construction of the equivalence classes see Fecher and Huth [6], Definition 10.

The set of transitions is calculated easily. Each OR-world is connected to its set of BRANCH-worlds and each BRANCH-world is connected to the OR-worlds encoded with itself. The labeling simply corresponds to the second part of each BRANCH-tuple.

Formally the μ-automaton $\widetilde{M}$ is defined as $\widetilde{M} = (O, B, \tilde{\imath}, \rightrightarrows, \rightarrow, L)$ such that

- $O = \{\tilde{\imath}, \tilde{\imath}', \tilde{\imath}''\} = \tilde{I}$,
- $B = \{\tilde{b}_1, \tilde{b}_2, \tilde{b}', \tilde{b}''\}$ with
  - $\tilde{b}_1 = (\{\tilde{\imath}, \tilde{\imath}'\}, (n > 0) \wedge \neg e)$,
  - $\tilde{b}_2 = (\{\tilde{\imath}, \tilde{\imath}''\}, (n > 0) \wedge e)$,
  - $\tilde{b}' = (\emptyset, (n = 0) \wedge \neg e)$,
  - $\tilde{b}'' = (\emptyset, (n = 0) \wedge e)\}$,
- $\rightrightarrows = \{(\tilde{\imath}, \tilde{b}_1), (\tilde{\imath}, \tilde{b}_2), (\tilde{\imath}', \tilde{b}'), (\tilde{\imath}'', \tilde{b}'')\}$,
- $\rightarrow = \{(\tilde{b}_1, \tilde{\imath}), (\tilde{b}_1, \tilde{\imath}'), (\tilde{b}_2, \tilde{\imath}), (\tilde{b}_2, \tilde{\imath}'')\}$, and
- $L(b) = \Pi_2(b)$ for all $b \in B$.

3.3 Generalized μ-Automata

The model of generalized μ-automata results from a combination of pre- and post-abstraction [18]. Its basic idea is to employ the simpleness of pre-abstraction, as far as possible, and to switch to post-abstraction if must-hypertransitions would lead to greater complexity. In other words one employs the generalized Kripke modal transition system where all must-hypertransitions are replaced by additional worlds and transitions according to a local post-abstraction.

Let $K = (W, \hat{w}, R, \mathcal{L})$ be a pointed Kripke structure, I a set of abstract worlds, $\alpha : I \to \overline{\mathcal{L}}$ an abstraction function, and $A = (Q, \hat{q}, \delta, \theta)$ an alternating tree automaton. In addition to the abstraction function, an *index function* ι is needed with $\iota : I \to \{\gamma, \mu\}$. This index function maps each abstract state $i \in I$ to the technique to be applied at that state: a local pre-abstraction (γ) or a local post-abstraction (μ).

Note. As an additional constraint, ι has to guarantee that $\iota(i) = \mu$ if local pre-abstraction at the abstract world i would lead to a must-hypertransition.

There are three disjunct sets of worlds in a generalized μ-automaton. The set of TRANS-worlds T and the respective transitions (starting in these worlds) represent the "pre-abstraction-part", thus the underlying pre-abstraction (i.e. a generalized Kripke modal transitions system) without must-hypertransitions. T consists of the abstract worlds $i \in I$ with $\iota(i) = \gamma$. The sets of OR-worlds O and BRANCH-worlds B and the respective transitions form the "post-abstraction-part" (i.e. a μ-automaton). O consists of the abstract worlds $i \in I$ with $\iota(i) = \mu$, while B consists of additional worlds, where each $b \in B$ is clearly assigned to one OR-world.

No actual means to construct a complete generalized μ-automaton for a given pointed Kripke structure with a set of abstract worlds, an abstraction function and an index function will be given here. Instead Section 3.3.1 Refinement Games on page 23 will present a technique to check whether a given generalized μ-automaton is an abstraction of a given pointed Kripke structure.

Definition 3.3.1 (Generalized μ-automaton)**.** A *generalized μ-automaton GM* for a pointed Kripke structure K is a tuple $(T, O, B, \hat{i}, R^-, R^+, \rightrightarrows, \to, L)$ such that

- T is the set of TRANS-worlds,
- O is the set of OR-worlds,
- B is the set of BRANCH-worlds,
- $\hat{i} \in (T \cup O)$ is its initial world,
- $R^-, R^+ \subseteq T \times (T \cup O)$ is the set of *must*- and *may*-transition relations (respectively),
- $\rightrightarrows \subseteq O \times B$ is the set of OR-transition relations,
- $\to \subseteq B \times (T \cup O)$ is the set of BRANCH-transition relations, and
- $L : (T \cup B) \to \overline{\mathcal{L}}$ is a labeling function.

GM is finite if T, O and B are.

Definition 3.3.2 (Well-formed)**.** A generalized μ-automaton GM is called *well-formed* iff $(w_1, w_2) \in R^-$ implies $(w_1, w_2) \in R^+$, thus $R^- \subseteq R^+$.

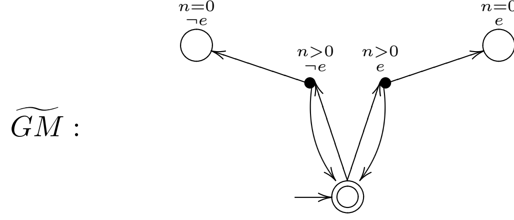

Figure 3.3: A *generalized μ-automaton.* TRANS-worlds ($\in T$) are depicted as unfilled circles, OR-worlds ($\in O$) as double-lined circles and BRANCH-worlds ($\in B$) as small solid circles. The labeling $L(w) \in \overline{\mathcal{L}}$ of a world w is depicted near the corresponding world. Solid arrows $t \to w$ between TRANS-states and from TRANS- to OR-states are must-transitions $(t, w) \in R^-$, while dashed arrows $t \dashrightarrow w$ are may-transitions $(t, w) \in R^+$. Arrows $o \to b$ denote OR-transitions $(o, b) \in \rightrightarrows$, while $b \to w$ denote BRANCH-transitions $(b, w) \in \to$. World names are omitted for the sake of simplicity.

Example 3.3.1. An example of a generalized μ-automaton resulting from the pointed Kripke structure in Example 2.2.1 on page 7 and the abstraction function $\tilde{\alpha}$ and the set of abstract worlds $\tilde{I}$ from Example 3.1.1 on page 16 is illustrated in Figure 3.3.

As one can easily see, the worlds of the generalized μ-automaton $\widetilde{GM}$ match the worlds of the generalized Kripke modal transition system $\widetilde{G}$ (Example 3.2.1 on page 18) almost perfectly. The only difference results from the world $\tilde{i}$ where the must-hypertransition in $\widetilde{G}$ starts. This world is replaced with $\tilde{i}$ from $\widetilde{M}$ (Example 3.2.2 on page 20) along with the BRANCH-worlds $\tilde{b}_1$ and $\tilde{b}_2$. Clearly, all worlds $w \in \{\tilde{i}', \tilde{i}''\}$ are inside the "pre-abstraction-part", while all worlds with $w \in \{\tilde{i}, \tilde{b}_1, \tilde{b}_2\}$ are inside the "post-abstraction-part".

For every world w from the "pre-abstraction-part" all transitions $\{w\}.R^+$ and $\{w\}.R^-$ from $\widetilde{G}$ are taken over to $\widetilde{GM}$ (none in this case). Similarly, for all worlds w from the "post-abstraction-part" all transitions $\{w\}.\rightrightarrows$ and $\{w\}.\to$ are taken over to their respective relations within $\widetilde{GM}$.

Finally the labeling function of each underlying automaton is wrapped to the respective worlds.

Formally the generalized μ-automaton $\widetilde{GM}$ is defined as:
$\widetilde{GM} = (T, O, B, \tilde{i}, R^-, R^+, \rightrightarrows, \to, L)$ with

- $T = \{\tilde{i}', \tilde{i}''\}$, $O = \{\tilde{i}\}$, $B = \{\tilde{b}_1, \tilde{b}_2\}$,
- $R^- = \{\}$,
- $R^+ = \{\}$,
- $\rightrightarrows = \{(\tilde{i}, \tilde{b}_1), (\tilde{i}, \tilde{b}_2)\}$,
- $\to = \{(\tilde{b}_1, \tilde{i}), (\tilde{b}_1, \tilde{i}'), (\tilde{b}_2, \tilde{i}), (\tilde{b}_2, \tilde{i}'')\}$, and
- $L \colon (T \cup B) \to \overline{\mathcal{L}}$ with $\tilde{i}' \mapsto (n = 0) \wedge \neg e; \tilde{i}'' \mapsto (n = 0) \wedge e;$
  $\tilde{b}_1 \mapsto (n > 0) \wedge \neg e; \tilde{b}_2 \mapsto (n > 0) \wedge e.$

3.3.1 Refinement Games

Generally, refinement is the opposite technique of abstraction. If model M_1 refines model M_2 then M_2 abstracts M_1. The notion of refinement games presented below, and developed in [18], allows one to correlate two given generalized μ-automata and construct an abstraction (refinement) hierarchy of such automata.

Definition 3.3.3 (Pointed Kripke structures as generalized μ-automata)**.** Pointed Kripke structures can be embedded in the set of generalized μ-automata. Given a pointed Kripke structure $K = (W, \hat{w}, R, \mathcal{L})$, one can construct an equivalent generalized μ-automaton $GM[K] = (W, \emptyset, \emptyset, \hat{w}, R, R, \emptyset, \emptyset, L)$, with

$$L : W \to \overline{\mathcal{L}};$$
$$w \mapsto \bigwedge_{p \in \mathcal{L} \wedge w \in [\![p]\!]}(p) \quad \wedge \quad \bigwedge_{p \in \mathcal{L} \wedge w \notin [\![p]\!]}(\neg p).$$

It is easy to see that for any alternating tree automaton A: $K \models A$ iff $GM[K] \models A$, since the property game for $GM[K]$ equals the property game for K (see Definition 3.3.7 on page 25).

Refinement of generalized μ-automata is defined with the help of *refinement games*, which are special kinds of parity games. Two players (*Player a* and *Player b*) play this game on a field consisting of two generalized μ-automata $GM_a = (T_a, O_a, B_a, \hat{w}_a, R_a^-, R_a^+, \rightrightarrows_a, \to_a, L_a)$ and $GM_b = (T_b, O_b, B_b, \hat{w}_b, R_b^-, R_b^+, \rightrightarrows_b, \to_b, L_b)$. To simplify the notations, define $W_a = T_a \cup O_a \cup B_a$ and $W_b = T_b \cup O_b \cup B_b$.

Each configuration of the game is a pair $(w_a, w_b) \in W_a \times W_b$. Initially a pawn is placed on the initial worlds of each automaton, resulting in the initial configuration $(\hat{w}_a, \hat{w}_b)$. Refinement games are played in rounds with a set of rules detailing which player moves the pawns in which way on the field each round, resulting in a new configuration. If more than one rule is applicable a single one is chosen by Player b. Each player wins a play iff the other one is losing. Winning conditions for finite plays are detailed by the rules. Additionally, a play is lost if one player is forced to take a turn which is impossible to do (e.g. one pawn should be moved to an unreachable/non-existing position). Infinite plays are won by Player a.

Definition 3.3.4 (Winning strategy)**.** A player is said to have a *winning strategy* iff he has a strategy to move the pawns on his turns that allows him to win all plays regardless of the moves the other player performs and the rules chosen.

Definition 3.3.5 (Refinement)**.** Let $GM_a = (T_a, O_a, B_a, \hat{w}_a, R_a^-, R_a^+, \rightrightarrows_a, \to_a, L_a)$ and $GM_b = (T_b, O_b, B_b, \hat{w}_b, R_b^-, R_b^+, \rightrightarrows_b, \to_b, L_b)$ be two generalized μ-automata. GM_a *refines* GM_b (and GM_b *abstracts* GM_a) iff Player a has a winning strategy for all refinement plays (using the rules stated in Definition 3.3.6 on the following page) between GM_a and GM_b started at $(\hat{w}_a, \hat{w}_b)$.

Definition 3.3.6 (Rules for refinement games)**.** The rules for refinement games between two generalized μ-automata $GM_a = (T_a, O_a, B_a, \hat{w}_a, R_a^-, R_a^+, \rightrightarrows_a, \rightarrow_a, L_a)$ and $GM_b = (T_b, O_b, B_b, \hat{w}_b, R_b^-, R_b^+, \rightrightarrows_b, \rightarrow_b, L_b)$ with $W_a = T_a \cup O_a \cup B_a$ and $W_b = T_b \cup O_b \cup B_b$ at configuration $(w_a, w_b) \in W_a \times W_b$ are presented below, divided by a case analysis on the considered configuration:

$w_a \in O_a$: Player b picks $w'_a \in \{w_a\}.\rightrightarrows$; next configuration: (w'_a, w_b),

$w_a \notin O_a \wedge w_b \in O_b$: Player a picks $w'_b \in \{w_b\}.\rightrightarrows$; next configuration: (w_a, w'_b),

$w_a \notin O_a \wedge w_b \notin O_b$: Player b chooses any of the following alternative rules:

1. Player a wins iff Satisfiable$(L_a(w_a) \wedge L_b(w_b))$
2. Player b picks $w'_b \in (\{w_b\}.R_b^-) \cup (\{w_b\}.\rightarrow_b)$;
   Player a picks $w'_a \in (\{w_a\}.R_a^-) \cup (\{w_a\}.\rightarrow_a)$;
   $\hookrightarrow$ the next configuration is (w'_a, w'_b)
3. Player b picks $w'_a \in (\{w_a\}.R_a^+) \cup (\{w_a\}.\rightarrow_a)$;
   Player a picks $w'_b \in (\{w_b\}.R_b^+) \cup (\{w_b\}.\rightarrow_b)$;
   $\hookrightarrow$ the next configuration is (w'_a, w'_b)

Refinement games are sequences of configurations generated by these rules.

Example 3.3.2. The generalized μ-automaton $\widetilde{GM}$ from Example 3.3.1 on page 22 is an abstraction of the pointed Kripke structure $\widetilde{K}$ from Example 2.2.1 on page 7, or more precisely an abstraction of the generalized μ-automaton $GM[\widetilde{K}]$.

So, GM_a is $GM[\widetilde{K}]$ and GM_b is $\widetilde{GM}$. The initial configuration is $(\tilde{w}_1, \tilde{i})$.

Player a picks $\tilde{b}_1$, resulting in configuration $(\tilde{w}_1, \tilde{b}_1)$. Player b may now pick one of three rules. If Player b picks the first rule, Player a would win.

If Player b picks the second rule and $w_b = \tilde{i}'$, Player a picks $w_a = \tilde{w}'$. Now, Player b can only select the first rule, and Player a wins.

If Player b picks the third rule and $w_a = \tilde{w}'$, Player a picks $w_b = \tilde{i}'$. Now, Player b can only select the first rule, and Player a wins.

If Player b picks the second rule and $w_b = \tilde{i}$, Player a picks $w_a = \tilde{w}_2$. If Player b picks the third rule and $w_a = \tilde{w}_2$, Player a picks $w_b = \tilde{i}$. Thus, the new configuration is $(\tilde{w}_2, \tilde{i})$, similar to the initial configuration.

A winning strategy for Player a is similar to the part sketched above. At a configuration $(\tilde{w}_{2n-1}, \tilde{i})$ for $n \in \mathbb{N}$, Player a picks $w_b = \tilde{b}_1$. At a configuration $(\tilde{w}_{2n}, \tilde{i})$ for $n \in \mathbb{N}$, Player a picks $w_b = \tilde{b}_2$. If Player b picks the second rule and $w_b = \tilde{i}'$, Player a picks $w_a = \tilde{w}'$. If Player b picks the second rule and $w_b = \tilde{i}''$, Player a picks $w_a = \tilde{w}''$. If Player b picks the third rule and $w_a = \tilde{w}'$, Player a picks $w_b = \tilde{i}'$. If Player b picks the third rule and $w_a = \tilde{w}''$, Player a picks $w_b = \tilde{i}''$. If Player b picks the second rule and $w_b = \tilde{i}$, Player a picks $w_a = \tilde{w}_{n+1}$. If Player b picks the third rule and $w_a = \tilde{w}_{n+1}$, Player a picks $w_b = \tilde{i}$.

It is easy to see that Player a wins all refinement plays if he utilizes this strategy. Thus $\widetilde{GM}$ abstracts $GM[\widetilde{K}]$.

3.3.2 Property Games

Satisfaction of a generalized μ-automaton with respect to a property encoded into an alternating tree automaton is obtained with the help of special strong-weak-parity-games, called *property games*.

Definition 3.3.7 (Property game)**.** A *property game* for a generalized μ-automaton $GM = (T, O, B, \hat{\imath}, R^-, R^+, \rightrightarrows, \rightarrow, L)$ and an alternating tree automaton $A = (Q, \hat{q}, \delta, \theta)$ is defined as the strong-weak-parity game $G_{GM,A} = (C_0, C_1, C_{0/1}, (\hat{\imath}, \hat{q}), G^-, G^+, G^\circ, \vartheta, \omega)$, where

- $C_0 = (T \cup B) \times Q_0$,
- $C_1 = (T \cup B) \times Q_1$,
- $C_{0/1} = (T \cup B) \times Q_{0/1} \bigcup O \times Q$,
- $G^- = \{((t,q),(i,q')) \mid \delta(q) \in \{\Diamond q', \Box q'\} \wedge (t,i) \in R^-\}$
  $\cup \{((b,q),(i,q')) \mid \delta(q) \in \{\Diamond q', \Box q'\} \wedge (b,i) \in \rightarrow\}$,
- $G^+ = \{((t,q),(i,q')) \mid \delta(q) \in \{\Diamond q', \Box q'\} \wedge (t,i) \in R^+\}$
  $\cup \{((b,q),(i,q')) \mid \delta(q) \in \{\Diamond q', \Box q'\} \wedge (b,i) \in \rightarrow\}$,
- $G^\circ = \{((i,q),(i,q')) \mid i \in (T \cup B) \wedge \exists q'' : \delta(q) \in \{q', q'\tilde{\vee} q'', q''\tilde{\vee} q', q'\tilde{\wedge} q'', q''\tilde{\wedge} q'\}\}$
  $\cup \{((o,q),(b,q)) \mid (o,b) \in \rightrightarrows\}$,
- $\vartheta(i,q) = \theta(q)$, and
- $\omega(i,q) = \begin{cases} tt, & \text{if } q \in Q_{0/1} \wedge i \in (T \cup B) \wedge (L(i) \Rightarrow \delta(q)) \\ f\!f, & \text{if } q \in Q_{0/1} \wedge i \in (T \cup B) \wedge (L(i) \Rightarrow \neg\delta(q)) \\ \bot, & \text{otherwise.} \end{cases}$

Definition 3.3.8 (Satisfaction for generalized μ-automata)**.** A generalized μ-automaton GM *satisfies* an alternating tree automaton A, written $GM \models A$ iff the corresponding property game $G_{GM,A}$ is valid.

This definition of satisfaction is consistent with the definition given in [18], albeit being more formalized.

Theorem 3.3.1. *A property game for a well-formed generalized μ-automaton is a well-formed strong-weak-parity-game.*

Proof. Let $M = (T, O, B, \hat{\imath}, R^-, R^+, \rightrightarrows, \rightarrow, L)$ be a well-formed generalized μ-automaton. Let A be an alternating tree automaton. Let $G_{M,A} = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ be the resulting strong-weak-parity-game.

According to Definition 2.4.6 on page 12, one has to show that $G^- \subseteq G^+$ in order to prove that $G_{M,A}$ is well-formed.

Each set of transitions (G^- and G^+) consists of the union of two distinct sets. The first set is dependant on R^- and R^+, respectively. Since M is well formed, the first set of G^- is a subset of the first set of G^+. The second set is only dependent on $\rightarrow$ in both cases, thus the second sets are the same. Hence, $G^- \subseteq G^+$. $\square$

Example 3.3.3. Considering the generalized μ-automaton $\widetilde{GM}$ in Example 3.3.1 on page 22 and the alternating tree automaton $\widetilde{A}$ in Example 2.3.1 on page 10, one can construct the associated property game $G_{\widetilde{GM},\widetilde{A}}$. This property game is illustrated in Figure 3.4 on the following page.

In the rest of this example I will use the less formal notion used in the figure to denote game positions (similar to Example 2.4.1 on page 14).

It is easy to find a winning strategy for Player 0 in this game:

- $\binom{n>0 \wedge e}{\Diamond_1}$: Player 0 picks $\binom{n=0 \wedge e}{\Box}$,
- $\binom{n>0 \wedge \neg e}{\Diamond_1}$: Player 0 picks $\binom{n=0 \wedge \neg e}{\Box}$,
- $\binom{n>0 \wedge e}{\Diamond_2}$: Player 0 picks $\binom{n>0}{\tilde{\wedge}}$, and
- $\binom{n>0 \wedge \neg e}{\Diamond_2}$: Player 0 picks $\binom{n>0}{\tilde{\wedge}}$.

The first two cases lead to an infinite game that is won by Player 0. The last two cases lead Player 1 being unable to move, thus Player 0 wins.

Its easy to see that the winning strategy is simply an adaptation of the winning strategy of Example 2.4.1 on page 14.

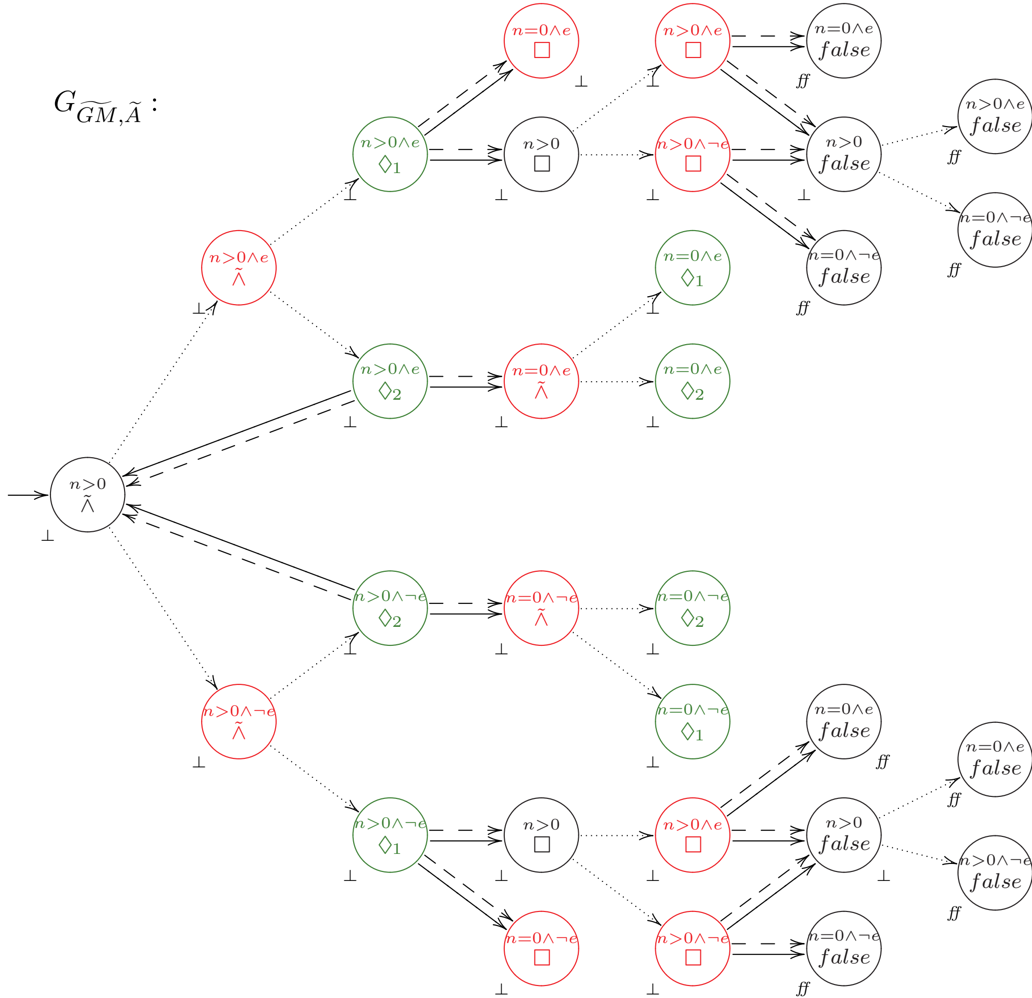

Figure 3.4: A *property game*. Green circles are C_0 game positions, red circles are C_1 game positions, and black circles are $C_{0/1}$ game positions. Values of the parity function ϑ are omitted (here always 0). Values of the validity function ω are depicted near the corresponding game positions. Solid arrows depict strong game transitions, dashed arrows weak game transitions, and dotted arrows junction game transitions. World names as well as state names are omitted, instead the predicates ($\in \overline{\mathcal{L}}$) and transition symbols ($\in \delta$) are depicted.

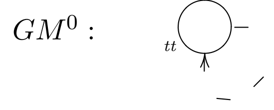

Figure 3.5: The *initial generalized μ-automaton.* The sole TRANS-world ($\in T$) is depicted as a unfilled circle. NO OR-worlds ($\in O$) or BRANCH-worlds ($\in B$) are present. The labeling $L(\Omega) \in \overline{\mathcal{L}}$ of the sole world Ω is depicted near the world. The dashed arrow $\Omega \dashrightarrow \Omega$ is a may-transitions $(\Omega, \Omega) \in R^+$. There are no must-, OR-, or BRANCH-transitions. World names are omitted for the sake of simplicity.

3.3.3 A Most Abstract Generalized μ-automaton

With the help of refinement games one can construct a generalized μ-automaton that is an abstraction of all possible pointed Kripke structures, called *initial generalized μ-automaton.*

Definition 3.3.9 (Initial generalized μ-automaton)**.** The *initial generalized μ-automaton* is defined as $GM^0 = (\{\Omega\}, \emptyset, \emptyset, \Omega, \emptyset, \{(\Omega, \Omega)\}, \emptyset, \emptyset, \Omega \mapsto \{tt\})$.

Thus the initial generalized μ-automaton consists of a single world. The sole may-transition means that in the concrete structure any transition may happen, but there need not be any transition. The initial generalized μ-automaton can be seen in Figure 3.5.

Lemma 3.3.1. *The initial generalized μ-automaton GM^0 abstracts any pointed Kripke structure.*

Proof. Let $K = (W, \hat{w}, R, \mathcal{L})$ be a pointed Kripke structure. Let $GM[K]$ be the associated generalized μ-automaton with $GM[K] = (W, \emptyset, \emptyset, \hat{w}, R, R, \emptyset, \emptyset, L)$.

Proving that $GM[K]$ refines GM^0 requires a winning strategy for Player a. Since there are no OR-worlds in $GM[K]$ or GM^0, only the set of three alternate rules will be used.

The first rule would always be a win for Player a, since Player b can only select tt, which holds true at every world of pointed Kripke structure. The second rule is not applicable since there are no must- or BRANCH-transitions in GM^0. The third rule lets Player b select any transition from the pointed Kripke structure. Player a simply selects the may-transition (Ω, Ω) and the play can continue. All infinite plays are won by Player a.

Thus Player a wins all plays and $GM[K]$ refines GM^0. $\square$

With the help of the initial generalized μ-automaton GM^0 one can construct an initial property game.

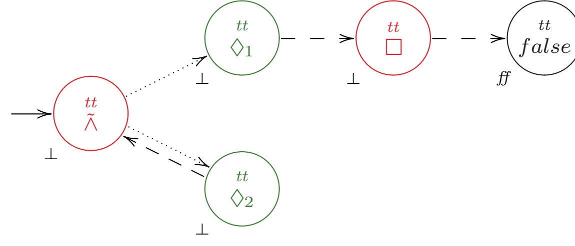

Figure 3.6: The *initial property game*. Green circles are C_0 game positions, red circles are C_1 game positions, and black circles are $C_{0/1}$ game positions. Values of the parity function ϑ are omitted (here always 0). Values of the validity function ω are depicted in small font. Dashed arrows depict weak game transitions, and dotted arrows depict junction game transitions. World names as well as state names are omitted, instead the predicates ($\in \overline{\mathcal{L}}$) and transition symbols ($\in \delta$) are depicted.

Definition 3.3.10 (Initial property-game)**.** The *initial property-game* for a pointed Kripke structure $K = (W, \hat{w}, R, \mathcal{L})$ and an alternating tree automaton $A = (Q, \hat{q}, \delta, \theta)$ is defined as the property-game $G_{GM^0,A}$ with
$G_{GM^0,A} = (\{\Omega\} \times Q_0, \{\Omega\} \times Q_1, \{\Omega\} \times Q_{0/1}, (\Omega, \hat{q}), \{\}, G^+, G^\circ, \vartheta, \omega)$, and

- $G^+ = \{((\Omega, q), (\Omega, q')) \mid \delta(q) \in \{\Diamond q', \Box q'\}\}$,
- $G^\circ = \{((\Omega, q), (\Omega, q')) \mid \exists q'' : \delta(q) \in \{q', q'\tilde{\vee}q'', q''\tilde{\vee}q', q'\tilde{\wedge}q'', q''\tilde{\wedge}q'\}\}$, and
- $\omega(\Omega, q) = \begin{cases} tt, & \text{if } q \in Q_{0/1} \wedge \delta(q) = tt \\ ff, & \text{if } q \in Q_{0/1} \wedge \delta(q) = ff \\ \bot, & \text{otherwise.} \end{cases}$

As expected this initial property game is independent of the pointed Kripke structure K. Furthermore the set of strong game transitions is empty. This means that this game is based upon a fully abstracted system, where any transition inside the the system may occur but does not have to.

Example 3.3.4. The initial property game $G_{Gm^0,\widetilde{A}}$ for the alternating tree automaton $\widetilde{A}$ from Example 2.3.1 on page 10 is illustrated in Figure 3.6.

If in a validity play Player 1 picks the path to configuration $\binom{tt}{\Diamond_1}$, Player 0 can't move, thus Player 1 wins all validity plays. Thus the game is not valid.

If in an invalidity play Player 1 picks the path to configuration $\binom{tt}{\Diamond_1}$, Player 0 can pick the path to $\binom{tt}{\Box}$. There, Player 1 can't move and looses the play. If Player 1 picks the path to configuration $\binom{tt}{\Diamond_2}$, Player 0 can pick the path back to $\binom{tt}{\tilde{\wedge}}$. Infinite plays are won by Player 0. Thus Player 0 wins all invalidity plays and the game is not invalid.

Thus initial property game $G_{Gm^0,\widetilde{A}}$ is neither valid nor invalid.

3.4 Abstract Property Games

The basic idea is that *abstract property games* are a superset of property games for generalized μ-automata that allow finer refinement steps.

The usual property game for generalized μ-automata is based upon a cross-product of the worlds of a generalized μ-automaton with the states of an alternating tree automaton. Any prior notion of refinement is based upon the generalized μ-automaton, thus any new refined property game will be a cross-product again. The algorithm presented in the next chapter will utilize a lazier notion of refinement. Instead of splitting any abstract world in the generalized μ-automaton, a single game position containing the abstract world will be split. Since any abstract world will be contained in more than one game position, but (usually) only one game-position will be split, the resulting new refined game is no longer the cross-product of a generalized μ-automaton with the alternating tree automaton. It is easy to see that there exists a series of refinement steps that will again lead to an usual property game (one simply splits all game positions that contain the abstract world).

Each abstract property game is seen in the context of a pointed Kripke structure $K = (W, \hat{w}, R, \mathcal{L})$ and an alternating tree automaton $A = (Q, \hat{q}, \delta, \theta)$ and consists of a tuple encoding a set of abstract worlds I for the pointed Kripke structure, an abstraction function α associating each abstract world with one or more concrete worlds, an index function ι assigning each abstract world to a local pre- or post-abstraction, and finally a strong-weak-parity-game $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ encoding the (in)validity of the abstract structure.

Similar to generalized μ-automata, each abstract world ($\in I$) is mapped to one of two sets (via ι), the set of TRANS-worlds T and the set of OR-worlds O. For each OR-world $o \in O$, there is a additional set of BRANCH-worlds B_o that is not part of I. The set of all BRANCH-worlds B is the union of all B_o for all $o \in O$. Each set B_o represents a split of the abstract world o into several finer abstract worlds ($\in B_o$), i.e. all BRANCH-worlds of the set B_o together describe the abstract world o. Each abstract world $b \in B_o$ encodes the abstract worlds reachable from b as well as a proposition taken from $\overline{\mathcal{L}}$ that is true at o as well as at all concrete worlds abstracted by b.

Similar to property games of generalized μ-automata, each game position ($c \in C$) of the strong-weak-parity-game G is a tuple $c = (i, q) \in (T \cup O \cup B) \times Q$. Furthermore game positions (i, q) containing abstract TRANS- and BRANCH-worlds as its first component are assigned to C_0, C_1, and $C_{0/1}$ according to $\delta(q)$. Game positions containing abstract OR-worlds as its first component are always assigned to $C_{0/1}$.

Strong and weak game transitions between two game positions (i, q) and (i', q') only occur if there is a possible change in the abstract world, i.e. the alternating tree automaton state contains a quantifier ($q \in Q_{quan}$) and the abstract world encoded in the first component is a TRANS- or BRANCH-world ($i \in T \cup B$). Junction game transitions only occur if the abstract world encoded in the first component stays the same, i.e. either the alternating tree automaton state contains no quantifier ($q \in Q_{\overline{quan}}$), or the game transition is a transition from an abstract OR-world to an abstract BRANCH-world ($i \in O$ and $i' \in B$).

Thus, similar to normal property games for generalized μ-automata, strong and weak

game transitions may only start in game positions containing TRANS- and BRANCH-worlds and end only in game positions containing TRANS- and OR-worlds. Furthermore, any transition starting in a game position containing an OR-world must be a junction game transition leading to a game position containing a BRANCH-world.

Definition 3.4.1 (Abstract property game)**.** An *abstract property game* for a pointed Kripke structure $K = (W, \hat{w}, R, \mathcal{L})$ and an alternating tree automaton $A = (Q, \hat{q}, \delta, \theta)$ is defined as the tuple $P_{K,A} = (I, \alpha, \iota, G)$ where I is a set of abstract states, α is an abstraction function, ι is an index function, and G is a strong-weak-parity-game with $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ such that

- $T = \{i \in I \mid \iota(i) = \gamma\}$,
- $O = \{i \in I \mid \iota(i) = \mu\}$,
- $B \subseteq \mathbb{P}(I) \times \overline{\mathcal{L}}$,

- $C_0 \subseteq (T \cup B) \times Q_0$,
- $C_1 \subseteq (T \cup B) \times Q_1$,
- $C_{0/1} \subseteq (T \cup B) \times Q_{0/1} \bigcup O \times Q$,

- $\exists i \in I$: $\hat{c} = (i, \hat{q}) \in C$,

- $G^-, G^+ \subseteq \{((i,q),(i',q')) \in (C_0 \cup C_1) \times C \mid i' \in T \cup O \wedge \delta(q) \in \{\Diamond q', \Box q'\}\}$,
- $G^\circ \subseteq \{((i,q),(i,q')) \mid i \in (T \cup B) \wedge \exists q'' : \delta(q) \in \{q', q' \tilde{\vee} q'', q'' \tilde{\vee} q', q' \tilde{\wedge} q'', q'' \tilde{\wedge} q'\}\}$
  $\cup \{((o,q),(b,q)) \mid o \in O \wedge b \in B\}$,

- $\vartheta(i,q) = \theta(q)$.

Definition 3.4.2 (Initial abstract property game)**.** The *initial abstract property game* for a pointed Kripke structure K and an alternating tree automaton A is the abstract property game $P^0_{K,A} = (I, \alpha, \iota, G)$ with

- $I = \{\Omega\}$,
- $\alpha(\Omega) = tt$,
- $\iota(\Omega) = \gamma$, and
- $G = G_{GM^0,A}$, the initial property game (Definition 3.3.10 on page 29).

Definition 3.4.3 (Validity)**.** An abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ is *valid* iff the strong-weak-parity-game G is valid. $P_{K,A}$ is *invalid* iff G is invalid.

Definition 3.4.4 (Soundness)**.** An abstract property game $P_{K,A}$ is said to be *sound* iff $P_{K,A}$ is valid implies that $K \models A$ and $P_{K,A}$ is invalid implies that $K \not\models A$.

Lemma 3.4.1 (Sound abstract property games). *Let $K = (W, \hat{w}, R, \mathcal{L})$ be a pointed Kripke structure and $A = (Q, \hat{q}, \delta, \theta)$ an alternating tree automaton. An abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ with $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ is sound if*

1. $\exists i \in I$: $\hat{c} = (i, \hat{q}) \in C \wedge \hat{w} \in [\![\alpha(i)]\!]$
   The abstract world i of the initial configuration $\hat{c}$ is an abstraction of the initial world $\hat{w}$.
2. *G is well-formed.*
3. $\forall (i, q), (i', q) \in C$ *with* $i, i' \in I : (i \neq i' \implies ([\![\alpha(i)]\!] \not\subseteq [\![\alpha(i')]\!] \wedge [\![\alpha(i')]\!] \not\subseteq [\![\alpha(i)]\!]))$
   There must not exist game positions that only differ in the first component and where one abstract world is a finer version of the other abstract world, i.e. i' abstracts a subset of the concrete worlds abstracted by i.
4. $\forall ((t, q), (i, q')) \in G^-$ *with* $t \in T$, $i \in I : \forall w \in [\![\alpha(t)]\!] : \exists w' \in [\![\alpha(i)]\!] : (w, w') \in R$
   Strong game transitions starting at a game position containing a TRANS-world relate to must transitions of generalized μ-automata and pre-abstraction respectively. Consequently for each strong game transition starting at a game position containing the abstract world t and ending at a game position containing i, there has to be a transition in the pointed Kripke structure for every w abstracted by t to any w' abstracted by i.
5. $\forall c \in C_0 \cup C_1$ *with* $c = (i, q)$ *and* $i \in I$ *and* $\omega(c) = \bot$:
   At all game positions that are not already valid or invalid...
   $$\delta(q) = \Diamond q' \implies (\forall w \in [\![\alpha(i)]\!].R : (w \models q') \implies \exists i' : ((i', q') \in C \wedge w \in [\![\alpha(i')]\!] \wedge ((i, q), (i', q')) \in G^+))$$
   ...if the second component of the game position is $\Diamond q'$, thus the game position is in C_0 and there is a possible change in the abstract world. Consider all concrete worlds w, reachable in one step from the concrete worlds abstracted by i. If the (concrete) property game $G_{K,A}$ is valid at game position (w, q') (short: $w \models q'$), then there has to be a game position (i', q') such that i' abstract w and there has to be a weak game transition from (i, q) to (i', q'). This rule ensures that Player 0 can disprove invalidity games.

   $$\delta(q) = \Box q' \implies (\forall w \in [\![\alpha(i)]\!].R : (w \not\models q') \implies \exists i' : ((i', q') \in C \wedge w \in [\![\alpha(i')]\!] \wedge ((i, q), (i', q')) \in G^+))$$
   ...if the second component of the game position is $\Box q'$, then the game position is in C_1 and there is a possible change in the abstract world. Consider all concrete worlds w, reachable in one step from the concrete worlds abstracted by i. If the (concrete) property game $G_{K,A}$ is invalid at game position (w, q') (short: $w \not\models q'$), then there has to be a game position (i', q') such that i' abstract w and there has to be a weak game transition from (i, q) to (i', q'). This rule ensures that Player 1 can disprove validity games.

   $$\delta(q) = q_1 \tilde{\vee} q_2 \implies (\forall j \in \{1, 2\} : ((\exists w \in [\![\alpha(i)]\!] : w \models q_j) \implies (i, q_j) \in \{c\}.G^\circ))$$
   ...if the second component of the game position is $q_1 \tilde{\vee} q_2$, then the game position is in C_0 and there is no change in the abstract world. Consider all concrete worlds

w abstracted by i. If the (concrete) property game $G_{K,A}$ is valid at game position (w, q_1) or (w, q_2), then there has to be a game position (i, q_1) or (i, q_2) respectively, that is reachable via a junction game transition from (i, q). This rule ensures that Player 0 can disprove invalidity games.

$\delta(q) = q_1 \tilde{\wedge} q_2 \implies (\forall j \in \{1,2\} : ((\exists w \in [\![\alpha(i)]\!] : w \not\models q_j) \implies (i, q_j) \in \{c\}.G^\circ))$
...if the second component of the game position is $q_1 \tilde{\wedge} q_2$, then the game position is in C_1 and there is no change in the abstract world. Consider all concrete worlds w abstracted by i. If the (concrete) property game $G_{K,A}$ is invalid at game position (w, q_1) or (w, q_2), then there has to be a game position (i, q_1) or (i, q_2) respectively, that is reachable via a junction game transition from (i, q). This rule ensures that Player 1 can disprove validity games.

6. *Let $Z_w = \{i \in I \mid \exists w' \in w.R : w' \in [\![\alpha(i)]\!]\}$ for all $w \in W$ be the set abstract worlds that abstract the concrete worlds that are reachable (within one step) from w.*
$\forall c \in C_{0/1}$ *with* $c = (o, q)$, $o \in \{i \in I \mid \iota(i) = \mu\} = O$, *and* $\omega(c) = \bot$:
For each game position that encodes an OR-worlds o as its first component and that is not yet valid or invalid...
$\forall Z \in \{\{w' \in [\![\alpha(o)]\!] \mid Z_{w'} = Z_w\} \mid w \in [\![\alpha(o)]\!]\}$:
$\exists c' \in ((\mathbb{P}(I) \times \overline{\mathcal{L}}) \times Q) \subseteq C$ *with* $c' = ((Z', P), q)$ *such that*
$$(\exists w \in Z : Z' = Z_w)$$
$$\wedge (P \implies \alpha(o)) \wedge (\forall w \in Z : \forall p \in \mathcal{L} : w \in [\![p]\!] \implies (p \implies P))$$
$$\wedge (c, c') \in G^\circ$$
$$\wedge (\omega(c') = tt \implies \forall w \in Z : w \models q) \wedge (\omega(c') = f\!f \implies \forall w \in Z : w \not\models q))$$
...for each equivalence class Z, there has to be a corresponding game position that encodes the abstract worlds that abstract any concrete world reachable (in one step) from any concrete world in the equivalence class in addition to the formula that holds true at all concrete worlds of the equivalence class. This equates to the definition of BRANCH-worlds in post-abstraction. The simplest formula for P is $P = \alpha(o)$. Furthermore each of these game positions c' containing an equivalence class is connected via a junction game transition to the game position c containing the respective OR-world.
Finally, if the validity function returns true or false at a game position (b, q), then all concrete worlds w contained in the equivalence class of b have to be (in)valid at (w, q).
7. $\forall c \in ((\mathbb{P}(I) \times \overline{\mathcal{L}}) \times Q) \subseteq C$ *with* $c = ((Z, P), q)$ *and* $\omega(c) = \bot$:
For each game position c encoding a BRANCH-world that is still unknown...
$$\delta(q) = \Diamond q' \implies (\forall i \in Z : \forall w \in [\![\alpha(i)]\!] : (w \models q')$$
$$\implies \exists c' \in C : (c' = (i, q') \wedge (c, c') \in G^+ \wedge (c, c') \in G^-))$$
...if the second component of the game position is $\Diamond q'$, then the game position is in C_0 and there is a possible change in the abstract world. Consider all concrete worlds w, abstracted by the abstract worlds encoded in Z, which are the abstract worlds reachable in one step from c. If the (concrete) property game $G_{K,A}$ is valid at game position (w, q') (short: $w \models q'$), then there has to be a game position (i, q') and there has to be a weak and a strong game transition from c to (i, q'). This rule

ensures that Player 0 can disprove invalidity games.

$\delta(q) = \Box q' \implies (\forall i \in Z : \forall w \in [\![\alpha(i)]\!] : (w \not\models q')$
$\implies \exists c' \in C : (c' = (i, q') \wedge (c, c') \in G^+ \wedge (c, c') \in G^-))$

...if the second component of the game position is $\Box q'$*, then the game position is in* C_1 *and there is a possible change in the abstract world. Consider all concrete worlds* w*, abstracted by the abstract worlds encoded in* Z*, which are the abstract worlds reachable in one step from* c*. If the (concrete) property game* $G_{K,A}$ *is invalid at game position* (w, q')*, then there has to be a game position* (i, q') *and there has to be a weak and a strong game transition from* c *to* (i, q')*. This rule ensures that Player 1 can disprove validity games.*

Let $W_{(Z,P)} = \{w \in W \mid \exists w' \in w.R : \exists i \in Z : (w' \in [\![\alpha(i)]\!] \wedge w' \in [\![P]\!])\}$ *be the set of concrete worlds abstracted by the BRANCH-world* (Z, P)*.*

$\delta(q) = q_1 \tilde{\vee} q_2 \implies (\forall j \in \{1, 2\} : ((\exists w \in W_{(Z,P)} : w \models q_j)$
$\implies \exists c' \in C : (c' = ((Z, P), q_j) \wedge (c, c') \in G^\circ))$

...if the second component of the game position is $q_1 \tilde{\vee} q_2$*, then the game position is in* C_0 *and there is no change in the abstract world. Consider all concrete worlds* w *abstracted by the BRANCH-world* (Z, P)*. If the (concrete) property game* $G_{K,A}$ *is valid at game position* (w, q')*, then there has to be a game position* $((Z, P), q')$ *and there has to be a junction game transition from* c *to* $((Z, P), q')$*. This rule ensures that Player 0 can disprove invalidity games.*

$\delta(q) = q_1 \tilde{\wedge} q_2 \implies (\forall j \in \{1, 2\} : ((\exists w \in W_{(Z,P)} : w \not\models q_j)$
$\implies \exists c' \in C : (c' = ((Z, P), q_j) \wedge (c, c') \in G^\circ))$

...if the second component of the game position is $q_1 \tilde{\wedge} q_2$*, then the game position is in* C_1 *and there is no change in the abstract world. Consider all concrete worlds* w *abstracted by the BRANCH-world* (Z, P)*. If the (concrete) property game* $G_{K,A}$ *is invalid at game position* (w, q')*, then there has to be a game position* $((Z, P), q')$ *and there has to be a junction game transition from* c *to* $((Z, P), q')$*. This rule ensures that Player 1 can disprove validity games.*

8. *Let* $W_{(Z,P)} = \{w \in W \mid \exists w' \in w.R : \exists i \in Z : (w' \in [\![\alpha(i)]\!] \wedge w' \in [\![P]\!])\}$ *be the set of concrete worlds abstracted by the BRANCH-world* (Z, P)*.*

   $\forall (((Z, P), q), (i, q')) \in G^-$ *with* $(Z, P) \in (\mathbb{P}(I) \times \overline{\mathcal{L}})$*,* $i \in I : \forall w \in W_{(Z,P)} : \exists w' \in [\![\alpha(i)]\!] : (w, w') \in R$
   Strong game transitions starting at a game position containing a BRANCH-world relate to BRANCH-transitions of generalized μ*-automata and post-abstraction respectively. Consequently for each strong game transition starting at a game position containing the abstract world* (Z, P) *and ending at a game position containing* i*, there has to be a transition in the pointed Kripke structure for every* w *abstracted by* (Z, P) *to any* w' *abstracted by* i*.*

9. $\forall c \in C$ *with* $c = (i, q)$ *and* $i \in I$ *it is:*
   $(\omega(c) = tt \implies \forall w \in [\![\alpha(i)]\!] : w \models q) \wedge (\omega(c) = ff \implies \forall w \in [\![\alpha(i)]\!] : w \not\models q)$
   If the validity function returns true or false at a game position (i, q) *then all concrete worlds* w *abstracted by* i *have to be valid or invalid at* (w, q)*.*

Proof. Let $P_{K,A}$ be a valid abstract property game that fulfils assertions 1.-9. Thus there exists a winning strategy for Player 0 to win all validity games regardless of moves Player 1 performs. Let P be the (concrete) property game of K and A. Let (w, q) be the current game position in the concrete property game and let (i, q) be the current game position in the abstract property game. Initially the current game position in both games is the respective initial game position. Assertion 1 ensures that the initial game position of the abstract property game corresponds to the initial game position of the concrete property game. (A concrete game position (w, q) corresponds to an abstract game position (i, q) if i abstracts w.)

Whenever Player 0 is able to make a move in the concrete property game, one has to deduce the strategy for the move from the winning strategy for the abstract property game. Whenever Player 1 makes a move in the concrete property game, one has to mirror the move in the abstract property game.

Since the abstract game is valid, no play will ever reach an invalid abstract game position. If the current abstract game position is valid, assertion 9 ensures that the concrete game is valid, too. If a play never reaches a valid abstract game position, i.e. the play is infinite, then the validness of the abstract property game ensures that the maximum of all infinitely often occurring parity numbers is even. The definition of abstract property games ensures that parity function only depends on the alternating tree automaton. The constructed winning strategy (and the mirroring of moves) sketched below, ensures that the abstract as well as the concrete property game always have the same second component in their respective game positions, thus the maximum of all infinitely often occurring parity numbers in the concrete property game is even, too. And thus P is valid, too.

Suppose $(w, q) \in C_0$, thus Player 0 must make a move. Since q is independent of the abstraction it follows that $(i, q) \in C_0$, too. Player 0 has a winning strategy for the abstract property game, thus he can select a new game position (i', q') that ensures that he will eventually win. There are two possibilities for the game transition $((i, q), (i', q'))$. The simplest one is $\delta(q) = q_1 \tilde{\vee} q_2$, so the game transition is a junction game transition (i.e. $i' = i$) and the definition of (concrete) property games ensures that there is a transition in P to the new game position (w, q') that corresponds to the new abstract game position (i, q'). The second possibility is $\delta(q) = \Diamond q'$. Thus the game transition is a strong game transition. Depending on $\iota(i)$ either assertion 4 or assertion 8 ensures that there exists a concrete game position (w', q') reachable in the concrete game that corresponds to (i', q').

Suppose $(w, q) \in C_1$ and Player 1 moves to (w', q'). This new game position can not be invalid, since assertion 5 and 7 would ensure that this move could be mirrored in the abstract property game which is valid. Thus the new game position (w', q') must be valid and there are only two possibilities: either there is no possible move in the abstract property game and thus it is valid too, or there is a possible move to a new game position (i', q') which corresponds to (w', q').

Whenever a move in the abstract property game reaches an OR game position, special care has to be applied. At these game positions Player 1 would perform an immediate extra move that is not immediately mirrored in the concrete property game. Thus one

moves the current abstract game position to the BRANCH game position that corresponds to (w', q'), as assured by assertion 6.

So, with the help of the winning strategy for the abstract property game, Player 0 is able to win all validity games in the concrete property game.

A similar reasoning holds true for invalid abstract property games that imply the invalidity of the corresponding concrete property games. □

Theorem 3.4.1. *The initial abstract property game $P^0_{K,A}$ is a sound abstract property game.*

Proof. It is evident that the initial abstract property game is an abstract property game. To prove its soundness all items of Lemma 3.4.1 on page 32 have to be shown.

1. Correct, since $\hat{c} = (\Omega, \hat{q})$ and $\hat{w} \in [\![\alpha(\Omega)]\!] = [\![tt]\!] = W$.
2. There are no strong game transitions, so G is well-formed.
3. There is only one abstract world.
4. There are no strong game transitions.
5. Definition 3.4.1 on page 31 ensures that at all game positions $(\Omega, q) \in C$ with $\delta(q) \in \{\Diamond q', \Box q'\}$ there is a weak game transition $((\Omega, q), (\Omega, q'))$. Thus, especially for all $w \in [\![\alpha(\Omega)]\!].R$ there is game position $(\Omega, q') \in C$ with $w \in [\![\alpha(\Omega)]\!]$ and $((\Omega, q), (\Omega, q'))$.
   Similarly Definition 3.4.1 ensures that at all game positions $(\Omega, q) \in C$ with $\delta(q) \in \{q_1 \tilde{\vee} q_2, q_1 \tilde{\wedge} q_2\}$ there are a junction game transitions $((\Omega, q), (\Omega, q_1))$ and $((\Omega, q), (\Omega, q_2))$.
6. There are no OR-worlds and no BRANCH-worlds.
7. There are no BRANCH-worlds.
8. There are no BRANCH-worlds.
9. For all $c \in C$ with $c = (\Omega, q)$ and $\omega(c) = tt$ one has $\delta(q) = tt$. Thus for all $w \in W$: $w \models tt$. And for all $c \in C$ with $c = (\Omega, q)$ and $\omega(c) = ff$ one has $\delta(q) = ff$. Thus for all $w \in W$: $w \not\models tt$.

Thus the initial abstract property game is sound. □

Chapter 4

CEGAR

4.1 Counter-Example Guided Abstraction Refinement

The usual approach to counter the state (world) explosion problem in model checking is abstraction. Gaining the actual abstraction, i.e. the abstraction function, usually requires considerable creativity and insight. Counter-example guided abstraction refinement, or short CEGAR, was invented by Clarke et al. [3] to enable the automatic generation of the abstraction.

The CEGAR-approach begins with an initial abstract model, that can be easily created. From this abstract model one constructs a property game to perform a validity play. This validity play either proves the property or provides a counterexample. Due to the abstraction the counterexample may be erroneous (or *spurious*). The next step is to perform an invalidity play on the property game. If this invalidity play confirms the counterexample, the property is disproved. Otherwise the counterexample is spurious and can be utilized to refine the abstract model. Then the cycle starts anew.

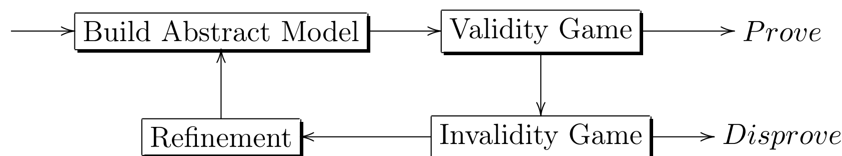

Figure 4.1: The *CEGAR-Cycle*. Typically this cycle runs until either the validity game proves the property or the invalidity game refutes it.

In the usual CEGAR-approach the abstraction function is modified in a way such that the abstract world responsible for the spurious counterexample is split into two more refined abstract worlds. Then a new abstract model is constructed with the help of the abstraction function.

Lazy abstraction, as defined by Henzinger et al. [14], works directly upon the abstract model. Instead of modifying the abstraction function and constructing a new abstract model from scratch, most of the abstract model is kept. Only the abstract world responsible for the spurious counterexample is split into two more refined abstract worlds and the transitions leading to or leaving the abstract world are adjusted.

Here an even lazier form of refinement is utilized. Instead of modifying the abstract model and constructing a new property game in each turn of the CEGAR-cycle, the game is modified directly.

Only the game position responsible for the spurious counterexample as well as all game positions connected via junction game transitions to this one have to be split. All of these game positions have in common that they contain the same abstract world as its first component. Thus an even finer version of refinement is utilized in order to avoid unnecessary blow up in the abstract structure.

This approach significantly shortens the CEGAR-cycle and reduces the size of the property games. There is no need to construct an abstract model in every turn, there is no need to construct a new property game in every turn, and only game positions that are really relevant to the spurious counterexample are split. In addition this approach allows a simplification of the games played. Validity or invalidity of parts of the game structure, which were not affected by changes due to the refinement, does not have to be rechecked, but remains known.

4.2 Simplified Games

Most strong-weak-parity-games can be simplified. For example, if any game position c with $\omega(c) = tt$ or $\omega(c) = \mathit{ff}$ is reached, no further transitions will be taken (due to the first rule of validity and invalidity games respectively). Thus any transitions leaving such a game position can be removed. Similarly at any game position $c \in C_0$ with $\omega(c) = \bot$, Player 0 has to choose a transition. If he chooses any transition leading to game position c' with $\omega(c') = \mathit{ff}$, he would lose the game. Likewise at any game position $c \in C_1$ with $\omega(c) = \bot$, Player 1 has to choose a transition. If he chooses any transition leading to game position c' with $\omega(c') = tt$, he would lose the game. Thus any of these transitions can be removed.

Definition 4.2.1 (Simplified game)**.** A strong-weak-parity-game G is called *simplified* if

(i) it is valid in $c \in C$ iff $\omega(c) = tt$,

(ii) it is invalid in $c \in C$ iff $\omega(c) = \mathit{ff}$, and

(iii) there are no game transitions (c, c') with

   a) $\omega(c) = tt$ or $\omega(c) = \mathit{ff}$,

   b) $c \in C_0$ and $\omega(c') = \mathit{ff}$, or

   c) $c \in C_1$ and $\omega(c') = tt$.

Theorem 4.2.1. *For any strong-weak-parity-game G there exists a simplified strong-weak-parity-game G' with*

- *G and G' utilize the same set of game positions C, and*
- *for all $c \in C$ it is: G is valid in c iff G' is valid in c.*
- *for all $c \in C$ it is: G is invalid in c iff G' is invalid in c.*

Algorithm `Simplify`(G)

1 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
2 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
3 Remove all game transitions (c, c') from G with $\omega(c) \neq \bot$.
4 Remove all game transitions (c, c') from G with $c \in C_0$ and $\omega(c') = ff$.
5 Remove all game transitions (c, c') from G with $c \in C_1$ and $\omega(c') = tt$.
6 `Return` the modified game G as G'.

Table 4.1: The algorithm `Simplify` gets a strong-weak-parity-game G as a parameter and returns a simplified strong-weak-parity-game G'.

Additionally the algorithm `Simplify` *(Table 4.1) calculates such a simplified G' for any strong-weak-parity-game G.*

Proof. Let $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ with $C = C_0 \cup C_1 \cup C_{0/1}$ be a strong-weak-parity-game. Let G' be the output of the algorithm `Simplify(G)` where ω' is the validity function of G'.

Since G is a strong-weak-parity-game and the only changes performed by the algorithm are the removal of some game transitions and the setting of $\omega'(c)$ to tt or ff for some $c \in C$, G' is a strong-weak-parity-game, too.

This also ensures the first point of the theorem. Let $c \in C$ be a valid game position in G. Line 1 of the algorithm ensures that $\omega'(c) = tt$. Thus any validity play on the game G' starting in c is won by Player 0. Thus G' is valid in c.

Let $c \in C$ be a valid game position in G'. Thus either $\omega'(c) = tt$ or there exists a winning strategy to win any validity play on G' starting in c. Assuming $\omega'(c) \neq tt$, it follows that G is not valid in c (otherwise line 1 of the algorithm would have set $\omega'(c) = tt$). Thus the removal of game transitions must have changed the validity in c. Line 3 removes game transitions that start in a game position $c' \in C$ with $\omega'(c') \neq \bot$. Any play that reaches such a c' ends, thus these transitions cannot influence validity. Line 4 removes game transitions (c', c'') with $\omega'(c') = \bot$ and $\omega'(c'') = ff$. Player 0 would not pick these transitions since they would mean him losing the play, additionally since $\omega'(c') = \bot$, there exists another game transition (c', c''') with $\omega'(c''') \neq ff$ (otherwise line 2 would have ensured $\omega'(c') = ff$). Thus these removed game transitions have no influence on validity. An analogous argument holds true for the transitions removed by line 5. Thus the assumption of $\omega'(c) \neq tt$ must be false. Further $\omega'(c) = tt$ iff either $\omega(c) = tt$ and thus c is valid in G or the algorithm set $\omega'(c) = tt$ and thus it is a valid game position in G. Equivalence of invalidity can be shown by similar means.

The last thing to prove is that G' is a simplified strong-weak-parity-game. (i) and (ii) are shown above. (iii) follows directly from the algorithm. □

Algorithm `PropertyCheck`(K, A)

1 Set the abstract property game $P_{K,A}$ to $P^0_{K,A}$
2 **while** $(\omega(\hat{c}) = \bot)$ **do**
3 `Simplify`(G)
4 Remove from G every game position $c \in C$ that is unreachable from $\hat{c}$ and every game transition leaving such a game position c.
5 `Refine`$(P_{K,A})$
6 **return** $\omega(\hat{c})$

Table 4.2: The algorithm `PropertyCheck` gets a pointed Kripke structure K and an alternating tree automaton A as parameters and returns the validity of the property encoded in A related to K.

4.3 Property Check

To check a given property encoded into an alternating tree automaton A with respect to a pointed Kripke structure K, one starts with the initial abstract property game $P^0_{K,A}$ (see Definition 3.4.2 on page 31). As long as this game is not determined in its initial game position, one refines the game.

Even an abstracted property game can become quite big. To reduce the size of the game one simplifies the strong-weak-parity-game (see Section 4.2 Simplified Games on page 38). Additionally, unreachable game positions can be removed. The complete algorithm `PropertyCheck` is presented in Table 4.2.

This algorithm utilizes $P_{K,A}$ as a local variable. $P_{K,A}$ always contains the current abstract property game (I, α, ι, G) with $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$, it is initialized with the initial abstract property game for the alternating tree automaton A.

As long as $P_{K,A}$ is not valid, the algorithm loops. First the abstract property game is simplified (`Simplify(`G`)`, see Table 4.1 on the previous page) and unreachable game positions are removed.

Finally a game position is selected and this game position, as well as all via junction game transitions reachable game positions, are refined (algorithm `Refine(`$P_{K,A}$`)`, see Table 4.3 on the following page). This mechanism is detailed in Section 4.4 Refinement on the next page.

Algorithm `Refine`($P_{K,A}$)

```
1 A Heuristic determines the used refinement approach.
2 % Heuristic determines split of game position c with predicate p with c = (i,q) and i ∈ I
3   Split(P_{K,A},c,p)
4 % Heuristic determines transformation of game position c with c = (i,q), i ∈ I, and ι(i) = γ
5   Transform(P_{K,A},c)
6 return the modified game P_{K,A}
```

Table 4.3: The algorithm `Refine` gets an abstract property game $P_{K,A}$ as a parameter and returns a refined version of the abstract property game.

4.4 Refinement

The algorithm `Refine` on Table 4.3 gets an abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ as an input and returns a modified abstract property game $P'_{K,A} = (I', \alpha', \iota', G')$. This new abstract property game is a refined version of the former one, i.e. one or more game positions are split or one or more game positions are transformed from local pre-abstraction to local post-abstraction. A heuristic determines which approach is used.

The first approach determines a game position $c \in C$ together with a predicate $p \in \overline{\mathcal{L}}$. This game position c consists of an abstract world $i \in I$ and an alternating tree automaton state $q \in Q$. This game position, as well as all game positions reachable via junction transitions will be split with the help of the predicate. An appropriate heuristic is employed to get the game position as well as the predicate.

The second approach determines a game position $c \in C$. This game position c consists of an abstract world $i \in I$ with $\iota(i) = \gamma$ and an alternating tree automaton state $q \in Q$. This game position, as well as all game positions reachable via junction transitions will be transformed. Thus $\iota(i)$ will be set to μ and corresponding game positions, consisting of BRANCH-worlds with the associated alternating tree automaton states, are created. An appropriate heuristic is employed to get the game position.

4.4.1 The Splitting Approach

The complete algorithm `Split` can be seen on pages 68–70. It gets an abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ with $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$, a game position $c = (i, q)$ with $i \in I$, and a predicate $p \in \overline{\mathcal{L}}$ as input and returns a modified abstract property game $P'_{K,A} = (I', \alpha', \iota', G')$.

Splitting the abstract world

The first step in the algorithm is to construct two new abstract worlds $i+$ and $i-$ from i. These two new abstract worlds should abstract subsets of the same concrete worlds as i, where $i+$ only abstracts the subset of concrete worlds w with $w \in [\![p]\!]$, thus $w \in [\![\alpha(i)]\!] \cap [\![p]\!] = [\![\alpha(i) \wedge p]\!]$. Similarly $i-$ only abstracts the subset of concrete worlds w with $w \notin [\![p]\!]$, thus $w \in [\![\alpha(i)]\!] \cap [\![\neg p]\!] = [\![\alpha(i) \wedge \neg p]\!]$.

So, one has to add such a new abstract world $i+$ with $[\![\alpha(i+)]\!] = [\![\alpha(i) \wedge p]\!]$ to the set of abstract worlds I and adjust the abstraction function α and the index function ι accordingly. $i-$ is treated similarly. It is entirely possible that this may lead to two (or more) abstract worlds i' and i'' with $\alpha(i') = \alpha(i'')$.

This is accomplished by the pseudo-code presented below:

Algorithm `Split`($P_{K,A}$,c,p)

```
1  % Splitting the abstract world
2  determine [[α(i) ∧ p]] and [[α(i) ∧ ¬p]]
3  add an i+ to I, adapt α so that α(i+) = (α(i) ∧ p), and adapt ι so that ι(i+) = ι(i).
4  add an i− to I, adapt α so that α(i−) = (α(i) ∧ ¬p), and adapt ι so that ι(i−) = ι(i).
```

Adding new game positions

First one has to determine all game positions that are connected to c via junction game transitions. Let C' be the set of these game positions. The definition of abstract property games (see Definition 3.4.1 on page 31) ensures that for all $c' \in C'$ there are only two possibilities: Either $c' = (i, q')$ for any $q' \in Q$ and $\iota(i) = \gamma$, i.e. the first component of these game positions is an abstract TRANS-world and they differ only in their second component, the alternating tree automaton state. Or $\iota(i) = \mu$, so all game positions $c' \in C'$ are either $c' = (i, q')$ for any $q' \in Q$ or $c' = (b, q')$ for any $q' \in Q$ and $b \in \mathbb{P}(I) \times \overline{\mathcal{L}}$, thus the first component of each such game position is either an abstract OR-world or an abstract BRANCH-world.

The following pseudo-code calculates the set C'. The local variable $C?$ contains all unchecked, connected (to c) game positions, initialized with c. As long as $C?$ contains any game positions, one stays within the while loop. Every checked game position (i', q') is removed from $C?$ and added to C'. Then, every game position $(i?, q?)$, reachable from (i', q') via any incoming or leaving junction game transition, is added to $C?$, but only if it was not checked before, i.e. $(i?, q?) \notin C'$.

```
5  % Calculate the set of game positions C'
6  C? := {(i, q)}; C' := ∅
7  while C? ≠ ∅ do
8    remove (i', q') from C?
9    C' := C' ∪ {(i', q')}
10   foreach (((i?, q?), (i', q')) ∈ G° or ((i', q'), (i?, q?)) ∈ G°) do
11     C? := C? ∪ {(i?, q?)} \ C'
```

The next step is the addition of the new game positions. For every $(i, q') \in C'$ two new, split game positions are needed: $(i+, q')$ and $(i-, q')$. For every new game position (i', q') combined with a parity value ϑ' and and a validity ω', the procedure `Add(`$P_{K,A}$`,` (i', q')`,` ϑ'`,` ω'`)` has to ensure that each new game position is added to the correct subset of C. For the new split game positions $(i+, q')$ and $(i-, q')$ the parity and validity values are inherited from the old game position (i, q').

Additionally one has to check whether the initial game position $\hat{c}$ is split, i.e. there exists $(i, \hat{q}) \in C'$ with $(i, \hat{q}) = \hat{c}$, that is $\hat{w} \in [\![\alpha(i)]\!]$. If this is the case, either $(i+, q')$ or $(i-, q')$ has to become the new initial game position.

Definition 2.2.4 on page 7 requires a formula $\hat{p}$ with $[\![\hat{p}]\!] = \{\hat{w}\}$. Thus with $\hat{w} \in [\![\alpha(i)]\!]$ one gets $[\![\hat{p}]\!] = [\![\hat{p}]\!] \cap [\![\alpha(i)]\!] = [\![\hat{p} \wedge \alpha(i)]\!]$. To calculate whether $\hat{w} \in [\![\alpha(i+)]\!]$, one simply determines if $[\![\hat{p}]\!] \cap [\![\alpha(i+)]\!] \neq \emptyset$. Thus $[\![\hat{p}]\!] \cap [\![\alpha(i+)]\!] = [\![\hat{p} \wedge \alpha(i+)]\!] = [\![\hat{p} \wedge \alpha(i) \wedge p]\!] = [\![\hat{p} \wedge \alpha(i)]\!] \cap [\![p]\!] = [\![\hat{p}]\!] \cap [\![p]\!] = [\![\hat{p} \wedge p]\!]$. Finally with $[\![false]\!] = \emptyset$, one can easily see, that call of `Satisfiable`$(\hat{p} \wedge p)$ is sufficient to determine whether $(i+, q')$ or $(i-, q')$ has to become the new initial game position.

```
12  % Add each new game position and adjust the initial game position
13  foreach (i,q') ∈ C' do
14    Add(P_{K,A}, (i+,q'),ϑ((i,q')), ω((i,q')))
15    Add(P_{K,A}, (i−,q'),ϑ((i,q')), ω((i,q')))
16    if ĉ = (i,q') then
17      if Satisfiable(p̂ ∧ p) then ĉ := (i+,q') else ĉ := (i−,q')
```

There are two cases to differentiate: $\iota(i) = \gamma$ and $\iota(i) = \mu$, thus whether the split occurs in the pre-abstraction part or in the post-abstraction part.

Splitting in the pre-abstraction part

```
18  % Split in the pre−abstraction part.
19  if ι(i) = γ then
```

In the first case, i.e. if $\iota(i) = \gamma$, there are three types of game transitions one has to check. The simplest part is checking each game position $c' \in C'$ for outgoing junction game transitions. It is to note, that there cannot be any incoming junction game transitions into any $c' \in C'$ which are not also outgoing junction game transitions originating from any $c'' \in C'$. This is due to C' including all game positions connected by junction game transitions.

The pseudo-code checks every game position $(i, q') \in C'$ that has junction game transitions. If there are any outgoing junction game transition in (i, q') they are removed and two new junction game transitions originating from the two new split game positions are added.

```
20  % Add new junction game transitions
21    foreach (i,q') ∈ C' do
22      foreach ((i,q'),(i,q?)) ∈ G° do
23        remove ((i,q'),(i,q?)) from G°
24        insert ((i+,q'),(i+,q?)) into G°
25        insert ((i−,q'),(i−,q?)) into G°
```

The next part is checking for outgoing and incoming weak game transitions. An outgoing weak game transition may occur at every game position $(i, q') \in C'$ with $q' \in Q_{Quan}$. If there is any outgoing weak game transition, one has to decide whether any of the two new split game position has to get an outgoing weak game transition.

Let (i', q') and (i'', q'') be two game positions. One has $((i', q'), (i'', q'')) \in G^+$ iff $q' \in Q_{quan}$ and $[\![\alpha(i')]\!] \cap [\![\text{pre}(\alpha(i''))]\!] \neq \emptyset$. Definition 2.2.2 on page 6 introduced the notation $[\![\text{pre}(\alpha(i''))]\!]$, that is $[\![\text{pre}(\alpha(i''))]\!]$ is the set of concrete worlds that have a world transition into a world abstracted by i''. Thus one gets a weak game transition, whenever there exists at least one of these concrete world that is also abstracted by i''. This is consistent

with the normal definition of may transitions (e.g. in the underlying generalized μ-automaton).

Similar to the determination of initial game positions, one can simplify the calculation of $[\![\alpha(i')]\!] \cap [\![\mathrm{pre}(\alpha(i''))]\!] \neq \emptyset$ to a satisfiability check.

```
26  % Add new outgoing weak game transitions
27    foreach (i, q′) ∈ C′ do
28      foreach ((i, q′), (i?, q?)) ∈ G⁺ do
29        remove ((i, q′), (i?, q?)) from G⁺
30        if Satisfiable(α(i+) ∧ pre(α(i?))) then insert ((i+, q′), (i?, q?)) into G⁺
31        if Satisfiable(α(i−) ∧ pre(α(i?))) then insert ((i−, q′), (i?, q?)) into G⁺
```

Incoming weak game transitions are calculated similarly:

```
32  % Add new incoming weak game transitions
33    foreach (i, q′) ∈ C′ do
34      foreach ((i?, q?), (i, q′)) ∈ G⁺ do
35        remove ((i?, q?), (i, q′)) from G⁺
36        if Satisfiable(α(i?) ∧ pre(α(i+))) then insert ((i?, q?), (i+, q′)) into G⁺
37        if Satisfiable(α(i?) ∧ pre(α(i−))) then insert ((i?, q?), (i−, q′)) into G⁺
```

The last part is checking for strong game transitions. An outgoing strong game transition $(c', c'') \in G^-$ may only occur whenever an outgoing weak game transition $(c', c'') \in G^+$ already exists. Thus all outgoing weak game transition starting in $(i+, q')$ and $(i-, q')$ are checked for outgoing strong game transitions.

If there already exists an outgoing strong game transition $((i, q'), (i?, q?)) \in G^-$ there will also be two strong game transitions $((i+, q'), (i?, q?))$ and $((i-, q'), (i?, q?))$. This corresponds to the usual notion of must transitions. An outgoing strong game transition $((i, q'), (i?, q?)) \in G^-$ exists if for every concrete world w abstracted by i there exists a concrete worlds w' abstracted by $i?$ with $(w, w') \in R$. Since $[\![\alpha(i+)]\!] \subseteq [\![\alpha(i)]\!]$ and $[\![\alpha(i-)]\!] \subseteq [\![\alpha(i)]\!]$, this also follows for $((i+, q'), (i?, q?))$ and $((i-, q'), (i?, q?))$.

If there is not already an existing outgoing strong game transition, one has to check if there should be one. $[\![\mathrm{pre}(\alpha(i?))]\!]$ are concrete worlds that have a transition to a concrete world abstracted by $i?$. Thus $[\![\neg\mathrm{pre}(\alpha(i?))]\!]$ are concrete worlds that have no transition to a concrete world abstracted by $i?$. So, if $[\![\alpha(i+)]\!] \cap [\![\neg\mathrm{pre}(\alpha(i?))]\!] = \emptyset$ there has to be a strong game transition. Similar with $i-$. This leads to the unsatisfiability checks seen in the pseudo-code below:

```
38  % Add new outgoing strong game transitions
39    foreach (i, q′) ∈ C′ do
40      foreach ((i+, q′), (i?, q?)) ∈ G⁺ do
41        if ((i, q′), (i?, q?)) ∈ G⁻ then
42          insert ((i+, q′), (i?, q?)) into G⁻
43        else
44          if not Satisfiable(α(i+) ∧ ¬pre(α(i?))) then insert ((i+, q′), (i?, q?)) into G⁻
45      foreach ((i−, q′), (i?, q?)) ∈ G⁺ do
46        if ((i, q′), (i?, q?)) ∈ G⁻ then
47          insert ((i−, q′), (i?, q?)) into G⁻
48        else
49          if not Satisfiable(α(i−) ∧ ¬pre(α(i?))) then insert ((i−, q′), (i?, q?)) into G⁻
50      foreach ((i, q′), (i?, q?)) ∈ G⁻ do
51        remove ((i, q′), (i?, q?)) from G⁻
```

The set of incoming strong game transitions is calculated differently. Since strong game transitions mostly depend on their starting point (there must *exists one* concrete world at the end point *for every* concrete world at the starting point), there will not be any new strong game transition added. Instead, existing incoming strong game transitions are checked and possibly made more precise, i.e. they are possibly assigned to one of the new game positions.

If there was an incoming strong game transition $((i?, q?), (i, q')) \in G^-$ which can not be assigned to any of the corresponding new game positions $(i+, q')$ or $(i-, q')$, there would have been a hypertransition. Thus, to avoid the loss of any information, the abstract world $i?$ has to be transformed into an OR-world.

```
52  % Add new incoming strong game transitions
53    foreach (i, q') ∈ C' do
54      foreach ((i?, q?), (i, q')) ∈ G⁻ do
55        remove ((i?, q?), (i, q')) from G⁻
56        added := false;
57        if ((i?, q?), (i+, q')) ∈ G⁺ then
58          if not Satisfiable(α(i?) ∧ ¬pre(α(i+))) then
59            insert ((i?, q?), (i+, q')) into G⁻
60            added := true;
61        if ((i?, q?), (i−, q')) ∈ G⁺ then
62          if not Satisfiable(α(i?) ∧ ¬pre(α(i−))) then
63            insert ((i?, q?), (i−, q')) into G⁻
64            added := true;
65        if not added then
66          Notify Heuristic of game position (i?, q?).
```

Splitting in the post-abstraction part

```
67  % Split in the post−abstraction part.
68  if ι(i) = μ then
```

In the second case, i.e. if $\iota(i) = \mu$, a lot more new game positions have to be added. More precisely, each BRANCH game position $((B, P), q') \in C'$ has to be split, too.

At each of these game positions $((B, P), q') \in C'$, B encodes all abstract worlds that abstract the concrete worlds reachable (within one step) from the concrete worlds abstracted by i, thus $B \in \mathbb{P}(I)$. The first step is to collect all these sets of abstract worlds in the set B'.

```
69  % Collect all BRANCH worlds in C'
70    B' := ∅
71    foreach ((B, P), q') ∈ C' do
72      B' := B' ∪ {B}
```

Each of these BRANCH-worlds will be split, similar to the split in the OR-world i, resulting in two new split BRANCH-worlds, $B+$ and $B-$ for every set in B'. The set of reachable abstract worlds for every new split abstract world $B+$ and $B-$ can only be a subset of B (due to being split). Thus every possible subset $I' \in \mathbb{P}B$ has to be tested until a matching subset is found.

$[\![\alpha(i+)]\!]$ encodes the concrete worlds abstracted by $i+$. Every abstract world i' that abstracts a concrete world reachable within one step from one of these worlds has to fulfil

$[\![\alpha(i+)]\!] \cap [\![\text{pre}(\alpha(i'))]\!] \neq \emptyset$. Thus to get all such abstract worlds, i.e. the set I', one has to check $[\![\alpha(i+)]\!] \cap \bigcap_{i' \in I'} [\![\text{pre}(\alpha(i'))]\!] \neq \emptyset$. Finally to get the maximal set I', one has to consider the abstract worlds not present in I'. An abstract world i' is not present in I' if $[\![\alpha(i+)]\!] \cap [\![\neg\text{pre}(\alpha(i'))]\!] \neq \emptyset$. This results in the final check for every possible subset $I' \in \mathbb{P}(B)$ of $[\![\alpha(i+)]\!] \cap \bigcap_{i' \in I'} [\![\text{pre}(\alpha(i'))]\!] \cap \bigcap_{i' \in B \setminus I'} [\![\neg\text{pre}(\alpha(i'))]\!] \neq \emptyset$.

This calculation ensures that there is only one possible I' that fulfils the check, thus as soon one is found no further checks are necessary. The algorithm presented below omits this aborting condition for better readability.

Finally for the second component of the BRANCH-world, the predicate ($\in \overline{\mathcal{L}}$) that holds true at all concrete worlds abstracted by $B+$, the algorithm uses the same calculation as for the satisfiability check. It is similar for $i-$ and $B-$.

The calculation of the check can again be performed as a satisfiability check, as presented below:

```
73  % Split each BRANCH world
74    foreach B ∈ B' do
75      foreach I' ∈ ℙ(B) do
76        φ+ := α(i+) ∧ ⋀_{i'∈I'} pre(α(i')) ∧ ⋀_{i'∈B\I'} ¬pre(α(i'))
77        φ− := α(i−) ∧ ⋀_{i'∈I'} pre(α(i')) ∧ ⋀_{i'∈B\I'} ¬pre(α(i'))
78        if Satisfiable(φ+) then B+ := (I', φ+)
79        if Satisfiable(φ−) then B− := (I', φ−)
```

The next step is the addition of the new split game positions. For each old BRANCH game position, simply two new split game positions are added.

```
80  % Add new BRANCH game positions
81    foreach ((B, P), q') ∈ C' do
82      Add(P_{K,A}, (B+, q'), ϑ(((B, P), q')), ω(((B, P), q')))
83      Add(P_{K,A}, (B−, q'), ϑ(((B, P), q')), ω(((B, P), q')))
```

Similar to the split in the pre-abstraction part, the transitions have to be adapted. The adaption of the junction transitions now consists of two parts. There are junction transitions connecting the OR game position with the BRANCH game position and junction game transitions connecting BRANCH game positions with other BRANCH game positions. The first one corresponds to the OR-transitions of generalized μ-automata and includes no transition of the second component. The second one corresponds to BRANCH-transitions where $\delta(q') \in Q_{\overline{quan}}$ and $\delta(q') \notin Q_{0/1}$.

The pseudo-code is similar to the one from the pre-abstraction part:

```
84  % Add new OR to BRANCH junction game transitions
85    foreach (i, q') ∈ C' with ∈ I do
86      foreach ((i, q'), ((B, P), q')) ∈ G° do
87        remove ((i, q'), (B, P), q')) from G°
88        insert  ((i+, q'), (B+, q')) into G°
89        insert  ((i−, q'), (B−, q')) into G°
90  % Add new BRANCH to BRANCH junction game transitions
91    foreach ((B, P), q') ∈ C' do
92      foreach (((B, P), q'), ((B, P), q?)) ∈ G° do
93        remove (((B, P), q'), (B, P), q?)) from G°
94        insert  ((B+, q'), (B+, q?)) into G°
95        insert  ((B−, q'), (B−, q?)) into G°
```

Outgoing weak and strong game transitions can only start in BRANCH game positions. Furthermore these transitions are always precise, i.e. if there exists a weak game transition there is a strong game transition, too.

The algorithm simply checks all outgoing game transitions and if the target is encoded in the first component of $B+$ or $B-$ respectively, a new game transition is added.

```
96   % Add new outgoing weak and strong game transitions
97     foreach ((B, P), q') ∈ C' do
98       foreach (((B, P), q'), (i?, q?)) ∈ G+ do
99         remove (((B, P), q'), (i?, q?)) from G+
100        remove (((B, P), q'), (i?, q?)) from G−
101        if i? ∈ Π1(B+) then
102          insert ((B+, q'), (i?, q?)) into G+
103          insert ((B+, q'), (i?, q?)) into G−
104        if i? ∈ Π1(B−) then
105          insert ((B−, q'), (i?, q?)) into G+
106          insert ((B−, q'), (i?, q?)) into G−
```

Incoming weak and strong game transitions can only end in OR game positions, thus the calculation is the same as in the pre-abstraction part.

```
107  % Add new incoming weak game transitions
108    foreach (i, q') ∈ C' with i ∈ I do
109      foreach ((i?, q?), (i, q')) ∈ G+ do
110        remove ((i?, q?), (i, q')) from G+
111        if Satisfiable(α(i?) ∧ pre(α(i+))) then insert ((i?, q?), (i+, q')) into G+
112        if Satisfiable(α(i?) ∧ pre(α(i−))) then insert ((i?, q?), (i−, q')) into G+
113  % Add new incoming strong game transitions
114    foreach (i, q') ∈ C' with i ∈ I do
115      foreach ((i?, q?), (i, q')) ∈ G− do
116        remove ((i?, q?), (i, q')) from G−
117        added := false;
118        if ((i?, q?), (i+, q')) ∈ G+ then
119          if not Satisfiable(α(i?) ∧ ¬pre(α(i+))) then
120            insert ((i?, q?), (i+, q')) into G−
121            added := true;
122        if ((i?, q?), (i−, q')) ∈ G+ then
123          if not Satisfiable(α(i?) ∧ ¬pre(α(i−))) then
124            insert ((i?, q?), (i−, q')) into G−
125            added := true;
126        if not added then
127          Notify Heuristic of game position (i?, q?).
```

Removing the old game positions

The final step in the algorithm is the removal of all old game positions that are now split. All incoming and outgoing game transitions from or to any of these old game positions are already removed by the algorithm.

```
128  % Add new incoming weak game transitions
129    foreach (i, q') ∈ C' do
130      remove (i, q') from P_{K,A} and adjust ϑ and γ
```

During the calculation of outgoing game transitions the old game positions may still be targeted. But every so created (now incoming) game transition is handled during the recalculation of the incoming transitions. Thus even self loops are handled well.

Furthermore, the splitting of the abstract world i into $i+$ and $i-$ produces new abstract worlds $i+$ and $i-$ every time it is performed. This ensures that (besides the most abstract world Ω) every via junction game transition connected part of the game consists of unique abstract game positions, i.e. in the transformation approach one can simply change ι of such an abstract world and only affect the small targeted portion of the game. The initial abstract property game could easily be modified to change Ω into several Ω_i, one for each via junction game transition connected part of the initial game. This reasonable (and if one wants to transform the abstract world Ω necessary) change has been omitted for an easier readability of the respective sections.

4.4.2 The Transforming Approach

The complete algorithm `Transform` can be seen on page 71. It gets an abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ with $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$, and a game position $c = (i, q)$ with $i \in I$ and $\iota(i) = \gamma$ as input and returns a modified abstract property game $P'_{K,A} = (I', \alpha', \iota', G')$.

Similar to the splitting approach, all via junction game transition connected game positions have to be transformed. The calculation of the set C' is similar to the calculation in the splitting approach.

Algorithm `Transform`($P_{K,A}$,c)

```
1  % Calculate the set of game positions C'
2  C? := {(i,q)}; C' := ∅
3  while C? ≠ ∅ do
4    remove (i,q') from C?
5    C' := C' ∪ {(i,q')}
6    foreach (((i?,q?),(i,q')) ∈ G° or ((i,q'),(i?,q?)) ∈ G°) do
7      C? := C? ∪ {(i?,q?)} \ C'
```

First, all TRANS game positions have to be transformed into OR game positions. All these game positions have their first component in common, the abstract world i. Thus the algorithm simply updates $\iota(i)$ to μ. As reasoned above, this change will only affect game positions in C'.

```
8  % Transform all TRANS game positions into OR game positions
9  ι(i) := μ
10 foreach c' ∈ C'
11   Move c' into C_{0/1}
```

The calculation of the abstract BRANCH-world belonging to the OR-world i, is similar to the calculation in the splitting approach. The first component of a BRANCH-world $B = (I', P)$ encodes all abstract worlds that abstract the concrete worlds reachable (within one step) from the concrete worlds abstracted by i, thus $I' \in \mathbb{P}(I)$. Every possible subset $I' \in \mathbb{P}(I)$ has to be tested until a matching subset is found.

$[\![\alpha(i)]\!]$ encodes the concrete worlds abstracted by i. Every abstract world i' that abstracts a concrete world reachable within one step from one of these worlds has to fulfil $[\![\alpha(i)]\!] \cap [\![\text{pre}(\alpha(i'))]\!] \neq \emptyset$. Thus to get all such abstract worlds, i.e. the set I', one has to check $[\![\alpha(i)]\!] \cap \bigcap_{i' \in I'} [\![\text{pre}(\alpha(i'))]\!] \neq \emptyset$. Finally to get the maximal set I', one has to consider the abstract worlds not present in I'. An abstract world i' is not present in I' if $[\![\alpha(i)]\!] \cap [\![\neg\text{pre}(\alpha(i'))]\!] \neq \emptyset$. This results in the final check for every possible subset $I' \in \mathbb{P}(I)$ of $[\![\alpha(i)]\!] \cap \bigcap_{i' \in I'} [\![\text{pre}(\alpha(i'))]\!] \cap \bigcap_{i' \in I \setminus I'} [\![\neg\text{pre}(\alpha(i'))]\!] \neq \emptyset$.

This calculation ensures that there is only one possible I' that fulfils the check, thus as soon as one is found no further checks are necessary. In the algorithm presented below this aborting condition is omitted for better readability.

Finally for the second component of the BRANCH-world, the predicate $(P \in \overline{\mathcal{L}})$ that holds true at all concrete worlds abstracted by B, the algorithm uses the same calculation as for the satisfiability check with the same reasoning.

The calculation of the check can again be performed as a satisfiability check, as presented below:

```
12  % Calculate BRANCH worlds
13  B' := ∅
14  foreach I' ∈ ℙ(I) do
15    φ := α(i) ∧ ⋀_{i'∈I'} pre(α(i')) ∧ ⋀_{i'∈I\I'} ¬pre(α(i'))
16    if Satisfiable(φ) then
17      B' := B' ∪ {(I', φ)}
```

The next step is the addition of the new BRANCH game positions. For every old TRANS game position the algorithm adds a new game position for every newly created BRANCH world.

```
18  % Add BRANCH game positions
19  foreach (i, q') ∈ C' do
20    foreach B ∈ B' do
21      Add(P_{K,A},(B, q'),ϑ((i, q')),ω((i, q')))
```

The adaption of the junction game transitions is straightforward. A junction game transition can only occur between two BRANCH game positions or connecting an OR game position to a BRANCH game position.

Every existing junction game transition is duplicated to each BRANCH game position created for the corresponding TRANS game positions. Then every OR game position is connected to each of its corresponding BRANCH game positions.

```
22  % Adapt junction game transitions
23  foreach (i, q') ∈ C' do
24    foreach ((i, q'), (i, q'')) ∈ G°
25      remove ((i, q'), (i, q'')) from G°
26      foreach B ∈ B' do
27        insert ((B, q'), (B, q'')) into G°
28  % Connect OR to BRANCH game positions
29  foreach (i, q') ∈ C' do
30    foreach B ∈ B' do
31      insert ((i, q'), (B, q')) into G°
```

Outgoing weak and strong game positions can only leave BRANCH game positions. Furthermore, whenever there is an outgoing weak game transition leaving a BRANCH game position, there is a strong game transition, too.

Thus, for every weak game transition leaving any of the old TRANS game positions the algorithm simply checks which of the corresponding BRANCH game positions encodes the correct target in its first component.

```
32  % Adapt outgoing weak and strong game transitions
33  foreach (i, q') ∈ C' do
34    foreach ((i, q'), (i?, q?)) ∈ G+
35      remove ((i, q'), (i?, q?)) from G+
36      remove ((i, q'), (i?, q?)) from G−
37      foreach B ∈ B' do
38        if i? ∈ Π1(B) then
39          insert  ((B, q'), (i?, q?)) into G+
40          insert  ((B, q'), (i?, q?)) into G−
```

Finally, there is no need to adapt any incoming weak or strong game transitions. Any of these can only end at an OR game position, and the switch of TRANS to OR does not change anything.

4.4.3 Adding new game positions

For every new game position (i, q) combined with a parity value ϑ' and and a validity ω', the procedure `Add(`$P_{K,A}$`, `(i,q)`, `ϑ'`, `ω'`)` ensures that each new game position is added to the correct subset of C.

Additionally, if $q \in Q_{0/1}$ in a TRANS- or BRANCH-world, the procedure has to ensure that the correct value of the validity function is calculated.

Procedure `Add`($P_{K,A}$,(i,q),ϑ', ω')

```
1  if i ∈ I ∧ ι(i) = μ % OR−world
2    C0/1 := C0/1 ∪ (i, q)
3  else % TRANS−world or BRANCH−world
4    if q ∈ Q0 then
5      C0 := C0 ∪ (i, q)
6    else if q ∈ Q1 then
7      C1 := C1 ∪ (i, q)
8    else % q ∈ Q0/1
9      C0/1 := C0/1 ∪ (i, q)
10     if α(i) ⇒ δ(q)
11       ω' := tt
12     else if α(i) ⇒ ¬δ(q)
13       ω' := ff
14     else
15       ω' := ⊥
16 % Adapt ϑ and ω
17 Set ϑ((i, q)) := ϑ'
18 Set ω((i, q)) := ω'
```

4.5 Soundness of the Algorithm PropertyCheck

To prove the soundness of the algorithm `PropertyCheck` (Table 4.2 on page 40) one has to ensure that the abstract property game $P_{K,A}$ that is utilized by the algorithm remains a sound abstract property game (Lemma 3.4.1 on page 32) after every step of the algorithm.

$P_{K,A}$ is initialized with the initial abstract property game $P^0_{K,A}$ which is a sound abstract property game (Theorem 3.4.1 on page 36).

Lemma 4.5.1. *Given a sound abstract property game* $P_{K,A} = (I, \alpha, \iota, G)$ *the abstract property game* $P'_{K,A} = (I, \alpha, \iota, G')$ *with* $G' = \texttt{Simplify}(G)$ *is a sound abstract property game.*

Proof. The algorithm `Simplify` (Table 4.1 on page 39) only modifies game transitions and the value of the validity function ω in a way such that game positions c with $\omega(c) = \bot$ become $\omega(c) = tt$ or $\omega(c) = ff$. Generally speaking, the algorithm only removes, but does not add anything. So, if any assertion was true for anything before the algorithm, it will still be true for anything after the algorithm. Only the interesting items of Lemma 3.4.1 will be detailed, the rest follows straightforwardly:

2. Whenever a weak game transition $(c, c') \in G^+$ is removed by the algorithm, a possible strong game transition $(c, c') \in G^-$ is removed, too. Thus the strong-weak-parity-game stays well-formed.
5. There are no new game positions $c \in C_0 \cup C_1$ with $c = (i, q)$ and $i \in I$ and $\omega(c) = \bot$. For any old game position c with these properties and with $\delta(q) = \Diamond q'$, i.e. $c \in C_0$: For all concrete worlds w, reachable in one step from the concrete worlds abstracted by i, if the (concrete) property game $G_{K,A}$ is valid at game position (w, q') (short: $w \models q'$), then there has to be a game position (i', q') such that i' abstracts w and there has to be a weak game transition from (i, q) to (i', q'). No game positions have been removed, so $c' = (i', q')$ still exists. In addition, since G is a sound game and there exists a w with $w \in [\![alpha(i')]\!]$ and $w' \models q$, c' can't be invalid, i.e. $\omega(c') \neq ff$. The weak game transition (c, c') would only have been removed if $c \in C_0$ and $\omega(c') = ff$, thus it still exists and the assertion holds true.

   A similar argument is also valid for any old game position c with $\delta(q) = \Box q'$, i.e. $c \in C_1$. Here, c' can not be valid, since there exists a w with $w \in [\![alpha(i')]\!]$ and $w' \not\models q$, thus $\omega(c) \neq tt$ and the weak game transition has not been removed.

   Finally similar arguments are valid for $\delta(q) = q_1 \tilde{\vee} q_2$ and $\delta(q) = q_1 \tilde{\wedge} q_2$.
6. There are no new game positions $c \in C_{0/1}$ with $c = (o, q)$ and $o \in O$ and $\omega(c) = \bot$. There are no game positions removed, thus all game positions containing BRANCH-worlds are still present. Furthermore there are no game transitions removed, that start in a game position $c \in C_{0/1}$ with $c \neq \bot$. Finally the algorithm will set $\omega(c') = tt$ iff c' is a valid game positions, i.e. for all $w \in Z : w \implies q$. It is similar for $\omega(c') = ff$.
7. This item is reasoned in an analog way to item 5.

9. If there are any new game positions $c \in C$ with $\omega(c) \neq \bot$, the assertion is assured by the first two lines of the algorithm.

Thus $P'_{K,A}$ is a sound abstract property game. $\square$

Lemma 4.5.2. *Given a sound abstract property game $P_{K,A} = (I, \alpha, \iota, G)$, the abstract property game $P'_{K,A} = (I, \alpha, \iota, G')$, where G' is the strong-weak-property-game G with all unreachable (from $\hat{c}$) game positions removed, is a sound abstract property game.*

Proof. Here, only unreachable game positions are removed, this may include game transitions between unreachable game positions. Like before, nothing is added to the game, thus if any assertion was true for each of anything, it remains true. Again, only interesting items will be detailed, the rest follows straightforwardly:

2. If any game transition leaving a game position is removed, all such game transitions are removed, thus the game remains well-formed.

5.-7. Since $P_{K,A}$ is a sound abstract property game, any required game position is connected, thus there will be nothing removed that is required by this assertion.

Thus $P'_{K,A}$ is a sound abstract property game. $\square$

Lemma 4.5.3. *Given a sound abstract property game $P_{K,A} = (I, \alpha, \iota, G)$, the abstract property game $P'_{K,A} = (I, \alpha, \iota, G')$ with $P'_{K,A} = \texttt{Split}(P_{K,A}, c, p)$ is a sound abstract property game.*

Proof. This lemma can straightforwardly be checked throughout the algorithm. In addition to details provided below most of the reasoning was already given during the description of the algorithm. The interesting items of Lemma 3.4.1 on page 32 are detailed below:

1. This is ensured by lines 16-17 of the algorithm, as detailed in the part of the algorithm on on page 43.
2. The algorithm only adds a strong game transition (c, c') if there already exists a weak game transition $(c, c') \in G^+$.
3. Each split game position ($\in C'$) is removed from the game in lines 129-130. For each of these removed game positions, two new game positions are added. Each of these pairs of new game positions contains the same abstract worlds $i+$ and $i-$. The calculation of these abstract worlds ensures that they abstract disjoint sets of concrete worlds, since $[\![\alpha(i) \wedge p]\!] = [\![\alpha(i)]\!] \cap [\![p]\!]$ and $[\![\alpha(i) \wedge \neg p]\!] = [\![\alpha(i)]\!] \cap W \setminus [\![p]\!]$.
4. Let $((t, q), (i', q')) \in G^-$ be a strong game transition added by the algorithm. Thus either t or i is a new abstract world. If t is one of the two new split abstract worlds then $((t, q), (i', q'))$ is a new outgoing strong game transition added in lines 38-51 of the algorithm. Suppose $t = i+$, thus either there was already a strong game transition $((i, q), (i', q'))$ or the unsatisfiability check holds true. In the first case the assertions holds true, since $[\![\alpha(i+)]\!] \subseteq [\![\alpha(i)]\!]$. In the second case suppose there exists a $w \in [\![\alpha(i+)]\!]$ such that for all $w' \in [\![\alpha(i')]\!]$ it is $(w, w') \notin R$, then

$[\![\alpha(i+)]\!] \cap \neg[\![\mathrm{pre}(\alpha(i'))]\!] \neq \emptyset$, thus the unsatisfiability check would be satisfiable and there would not have such a strong game transition been added. The same reasoning holds true for $t = i-$.

If i' is one of the new split abstract worlds then $((t, q), (i', q'))$ is a new incoming strong game transition added either in lines 52-66 or in lines 113-127 of the algorithm. In both cases a similar reasoning to above holds true.

5. Since the assertion holds true before the algorithm runs, the algorithm has to assure that there still exists any needed junction or weak game transition that involves a split game position after the algorithm ran. This is ensured by lines 20-37 of the algorithm and the reasoning is included in the description of these lines.
6. As reasoned in the description of lines 73-79 of the algorithm the calculated set of BRANCH-worlds corresponds to the set of equivalence classes in the assertion. For each equivalence class the algorithm calculates a BRANCH-world that encodes the set Z' of reachable abstract worlds. Furthermore, $\phi+$ respectively $\phi-$ corresponds to P of the assertion. The set of junction game transitions is added in lines 84-89. The last assertion of this item is ensured by the inheritance of ω and the procedure `Add`.
7. A similar reasoning to item 5 holds true.
8. A similar reasoning to item 4 holds true.
9. This assertion is ensured by the inheritance of ω and the procedure `Add`.

□

Lemma 4.5.4. *Given a sound abstract property game $P_{K,A} = (I, \alpha, \iota, G)$ the abstract property game $P'_{K,A} = (I, \alpha, \iota, G')$ with $P'_{K,A} =$ `Transform`$(P_{K,A}, c)$ is a sound abstract property game.*

Proof. This lemma can straightforwardly be checked throughout the algorithm. In addition to details provided below most of the reasoning was already given during the description of the algorithm. The interesting items of Lemma 3.4.1 on page 32 are detailed below:

2. The algorithm always adds a strong and a weak game transition.
6. This can be reasoned similarly to Lemma 4.5.3, item 6.
7. This follows from item 6 of the assertion and how the algorithm transforms these game transitions to the new BRANCH game transitions.
8. This can be reasoned similarly to Lemma 4.5.3, item 4.

□

Theorem 4.5.1 (Soundness of PropertyCheck)**.** *The algorithm `PropertyCheck` is sound for any pointed Kripke structure K and any alternating tree automaton A, i.e. if the algorithm return tt: $K \models A$ and if the algorithm returns ff: $K \not\models A$.*

Proof. This theorem follows directly from $P^0_{K,A}$ being a a sound abstract property game and Lemmata 4.5.1, 4.5.2, 4.5.3 and 4.5.4. □

4.6 Heuristics

The CEGAR-based algorithm `PropertyCheck` uses a heuristic in its `Refinement` to decide whether `Split` or `Transform` is used for the local refinement step. Additionally the heuristic determines the affected game state c and a potential predicate p, needed for the split.

Let $P = (I, \alpha, \iota, G)$ be an abstract property game with the strong-weak-parity-game $G = (C_0, C_1, C_{0/1}, \hat{c}, G^-, G^+, G^\circ, \vartheta, \omega)$ and $C = C_0 \cup C_1 \cup C_{0/1}$.

Definition 4.6.1 (Predicate-unknown)**.** A game position $c \in C$ with $c = (i, q)$ is *predicate-unknown* if $q \in Q_{0/1}$ and $\omega(c) = \bot$.

Definition 4.6.2 (Real may transition)**.** A *real may transition* is a weak game transition $t \in G^+$ that has no corresponding strong game transition, i.e. if $t = (c, c') \in G^+$ then there exists no corresponding strong game transition $(c, c') \in G^-$.

Predicate unknown game states as well as game states with an outgoing real may transition are good candidates for a split. Both of these kinds of game states can be seen as causes of uncertainty.

Let c be a predicate unknown game position, i.e. $c \in C_{0/1}$ and there can not be any outgoing game transition starting in c. Thus in any validity play that reaches c, there is no possible further move and Player 1 wins the play. Additionally, in any invalidity play that reaches c, there is no possible further move and Player 0 wins the play. So neither Player 0 can win a validity play that reaches c nor can Player 1 win any invalidity play at c.

Let c be a game position with an outgoing real may transition leading to c'. If $c \in C_0$ then Player 0 can possibly not reach c' , in any validity play and thus possibly looses the play at game position c. Furthermore in any invalidity play, Player 0 can reach c' and thus can possibly prevent Player 1 from winning. Thus it is possible that neither Player 0 can win any validity play nor can Player 1 win any invalidity play due to the real may transition. A similar argument holds for $c \in C_1$.

Thus possible strategies for the heuristic are:

- Determine a predicate unknown game state $c = (i, q)$ and order a split of c with $p = \delta(q)$.
- Determine a real may transition (c, c') with $c = (i, q)$ and $c' = (i', q')$ and order a split of c with $p = \mathrm{pre}(\alpha(i'))$.

With the first strategy the algorithm `Split` will split at least the predicate unknown game position (i, q) into $(i+, q)$ and $(i-, q)$, with $\alpha(i+) = (\alpha(i) \wedge p)$ and $\alpha(i-) = (\alpha(i) \wedge \neg p)$. The procedure `Add` ensures that $\omega((i+, q)) = tt$ and $\omega((i-, q)) = f\!f$, since $(\alpha(i+) \Rightarrow \delta(q)) = ((\alpha(i) \wedge p) \Rightarrow p) = true$ and $(\alpha(i-) \Rightarrow \neg\delta(q)) = ((\alpha(i) \wedge \neg p) \Rightarrow \neg p) = true$. Thus after the the split there is one cause of possible uncertainty removed: the predicate unknown game position c.

With the second strategy the algorithm `Split` will split at least the game position $c = (i, q)$ with the outgoing real may transition into two new game positions $(i+, q)$ and $(i-, q)$. Of these two new game positions only $(i+, q)$ may have an outgoing weak game transition to c', since $\text{SAT}(\alpha(i-) \wedge \text{pre}(\alpha(i'))) = \text{SAT}(\alpha(i) \wedge \neg\text{pre}(\alpha(i')) \wedge \text{pre}(\alpha(i'))) = false$. If there is a new outgoing weak game transition $((i+, q), (i', q')) \in G^+$ then there will also be an outgoing strong game transition $((i+, q), (i', q')) \in G^+$, since $\neg\text{SAT}(\alpha(i+) \wedge \neg\text{pre}(\alpha(i'))) = \neg\text{SAT}(\alpha(i) \wedge \text{pre}(\alpha(i')) \wedge \neg\text{pre}(\alpha(i'))) = \neg\text{SAT}(\alpha(i) \wedge false) = true$. Thus after the the split there is one cause of possible uncertainty removed: the real may transition (c, c').

Furthermore the algorithm `Split` is able to inform the heuristic of a possible loss of precision, i.e. before the split there was a strong game transition (c, c') that is lost after the split, i.e. there is neither a strong game transition $(c, c+)$ nor $(c, c-)$, where $c+$ and $c-$ are the two new game positions resulting of the split of c'.

If the heuristic is notified of this condition a possible strategy to get back the lost precision is:

- Transform the game position c.

Due to the construction of the BRANCH game positions there will be a strong game transition for each weak game transition leaving a BRANCH game position. Since there was a strong game transition (c, c') before the split, there was at least one weak game transition after the split, i.e. $(c, c+)$ or $(c, c') \in G^+$. Thus the new BRANCH game position corresponding to c will have a strong game transition to $c+$ or to $c-$.

Algorithm `OptimizedPropertyCheck`(K, A)

1 Set the abstract property game $P_{K,A}$ to $P^0_{K,A}$
2 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
3 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
4 **while** $(\omega(\hat{c}) = \bot)$ **do**
5 Remove all game transitions (c, c') from G with $\omega(c) \neq \bot$.
6 Remove all game transitions (c, c') from G with $c \in C_0$ and $\omega(c') = ff$.
7 Remove all game transitions (c, c') from G with $c \in C_1$ and $\omega(c') = tt$.
8 Remove from G every game position $c \in C$ that is unreachable from $\hat{c}$ and every game transition leaving such a game position c.
9 A Heuristic determines the used refinement approach.
10 *% Heuristic determines split of game position c with predicate p with $c = (i, q)$ and $i \in I$*
11 `Split`$(P_{K,A}$,c,$p)$
12 *% Heuristic determines transformation of game position c with $c = (i, q)$, $i \in I$, and $\iota(i) = \gamma$*
13 `Transform`$(P_{K,A}$,$c)$
14 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
15 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
16 **return** $\omega(\hat{c})$

Table 4.4: The algorithm `OptimizedPropertyCheck` gets a pointed Kripke structure K and an alternating tree automaton A as parameters and returns the validity of the property encoded in A related to K.

4.7 Optimization

The algorithm `PropertyCheck` can easily be optimized to reduce unneeded calculations. The optimized algorithm `OptimizedPropertyCheck` is presented in Table 4.4.

The only change to the algorithm `PropertyCheck` from Table 4.2 on page 40 is the direct inclusion of `Simplify` fromTable 4.1 on page 39 into the algorithm and the movement of the breaking condition after the determination of (in)valid game positions. Thus if the game is valid or invalid no further (unneeded) calculations are performed.

It is easy to see that the optimized algorithm still retains soundness.

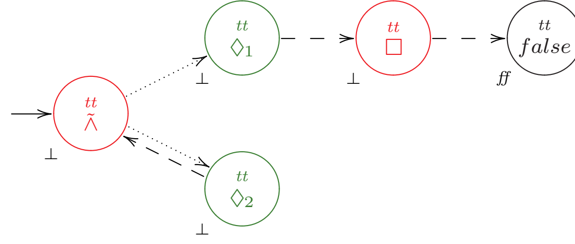

Figure 4.2: The *initial abstract property game*. Throughout the example, green circles are C_0 game positions, red circles are C_1 game positions, and black circles are $C_{0/1}$ game positions. Values of the parity function ϑ are omitted (here always 0). Values of the validity function ω are depicted in small font. Dashed arrows depict weak game transitions, solid arrows depict strong game transitions, and dotted arrows depict junction game transitions. World names of TRANS- and OR-worlds as well as state names are omitted, instead the predicates ($\in \overline{\mathcal{L}}$) and transition symbols ($\in \delta$) are depicted. Worlds names of BRANCH-worlds will be included instead of the predicates.

4.8 Example

This example will demonstrate the algorithm `PropertyCheck` (or rather the algorithm `OptimizedPropertyCheck`) with the help of the sample pointed Kripke structure $\widetilde{K}$ from Example 2.2.1 on page 7 and the sample alternating tree automaton $\widetilde{A}$ from Example 2.3.1 on page 10. Note that the heuristic in this example is not limited to the strategies described in Section 4.6 Heuristics on page 54.

The abstract property game of the algorithm gets initialized with the initial abstract property game from Definition 3.4.2 on page 31 and Example 3.3.4 on page 29. The most important part of the abstract property game (the strong-weak-parity-game) can be seen in Figure 4.2.

Since validity at the initial game position is still undecided, the first turn of the algorithm starts. Neither the validity play not the invalidity play results in any new information, thus there is nothing to simplify.

Thus the next step in the algorithm is refinement. There are several causes why neither Player 0 can win the validity game nor Player 1 can win the invalidity game, e.g. the weak game transition leaving $\binom{tt}{\diamond_2}$ causes Player 0 to loose all validity plays at that game position and prevents Player 1 from winning all invalidity plays at that game position. Similarly, the weak game transition leaving $\binom{tt}{\square}$ causes Player 1 to loose all invalidity plays at that game position and prevents Player 0 from winning all validity plays at that game position.

In this example the *heuristic* selects the second cause sketched above, thus a split of the game position $\binom{tt}{\square}$. The predicate selected by the heuristic is $e \in \overline{\mathcal{L}}$.

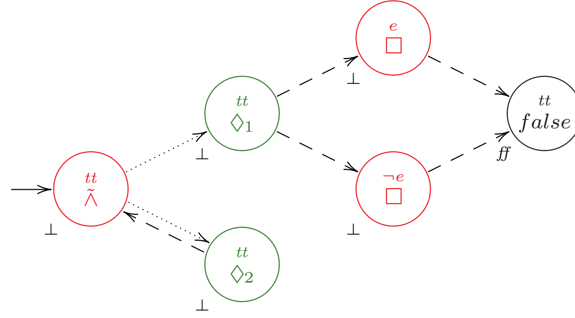

Figure 4.3: The *first turn: Refinement.*

The algorithm `Split` first calculates $i+$ and $i-$ with $\alpha(i+) = tt \wedge e = e$ and $\alpha(i-) = \neg e$. The set of connected game positions C' only consists of the sole game position $\binom{tt}{\Box}$, thus only two new game positions are added: $\binom{e}{\Box}$ and $\binom{\neg e}{\Box}$.

Since there is only one game position in C', there are no junction game transitions to check. The algorithm thus begins with outgoing weak game transitions. There is a single one, thus two satisfiability checks have to be performed. $\text{SAT}(\alpha(i+) \wedge \text{pre}(tt))$ and $\text{SAT}(\alpha(i-) \wedge \text{pre}(tt))$. Example 2.2.1 on page 7 gives $\text{pre}(tt) = (n > 0)$, thus $\text{SAT}(e \wedge (n > 0))$ and $\text{SAT}(\neg e \wedge (n > 0))$. Both checks are satisfiable, so both junction game transitions are added.

Incoming weak game transitions are handled in a similar way. The algorithm has to check $\text{SAT}(tt \wedge \text{pre}(e))$ as well as $\text{SAT}(tt \wedge \text{pre}(\neg e))$. With $\text{pre}(e) = (n > 0)$ and $\text{pre}(\neg e) = (n > 0)$ one gets $\text{SAT}(n > 0)$ in both cases, thus two new incoming weak game transitions are added.

The next step is the calculation of outgoing strong game transitions. Each newly added outgoing weak game transition is checked for a possible strong one. Thus two (un)satisfiability checks are performed: $\neg\text{SAT}(e \wedge \neg\text{pre}(tt))$ and $\neg\text{SAT}(\neg e \wedge \neg\text{pre}(tt))$. Thus $\neg\text{SAT}(e \wedge (n = 0))$ and $\neg\text{SAT}(\neg e \wedge (n = 0))$, both are satisfiable and no outgoing strong game transitions are added. Furthermore there are no incoming strong game transitions and the algorithm `Split` finishes with the removal of the old game position $\binom{tt}{\Box}$.

The resulting strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.3.

A new validity play as well as a new invalidity play results in no new information, thus again nothing can be simplified and the second turn of the algorithm starts.

This time the *heuristic* selects the first cause sketched above, thus a split of the game position $\binom{tt}{\Diamond_2}$. The predicate selected by the heuristic is $(n = 0) \in \overline{\mathcal{L}}$.

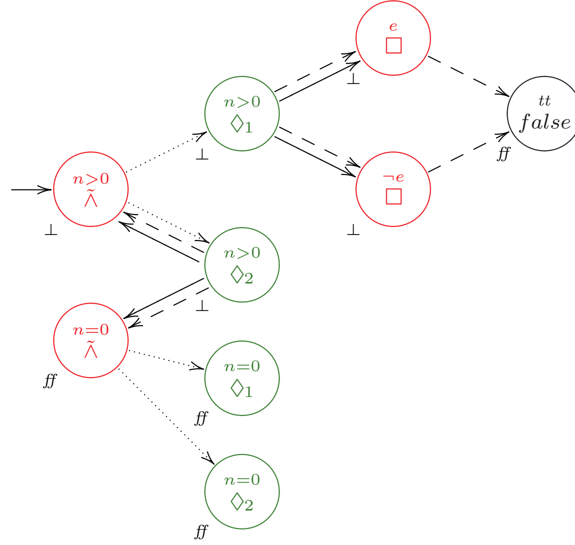

Figure 4.4: The *second turn: Refinement.*

Again, the algorithm `Split` determines the new abstract TRANS-worlds $i+$ and $i-$ with $\alpha(i+) = (n = 0)$ and $\alpha(i-) = (n > 0)$. The set C' of via junction game transition connected game positions contains three different game positions this time: $C' = \left\{ \binom{tt}{\tilde{\wedge}}, \binom{tt}{\Diamond_1}, \binom{tt}{\Diamond_2} \right\}$.

Hence six new game positions are added, and since two of them may become the new initial game position, the unsatisfiability of a satisfiability check of $\mathrm{SAT}(\hat{p} \wedge (n = 0)) = \mathrm{SAT}((n = 1) \wedge e \wedge (n = 0))$ determines that $\binom{n>0}{\tilde{\wedge}}$ becomes the new initial game position.

The set of new junction game transitions is calculated straightforwardly. Each existing junction game transition between two game positions of the set C' is mirrored to the respective game positions containing $i+$ and $i-$.

For each existing outgoing weak game transition, there have to be two satisfiability checks performed, one check each to decide whether the outgoing game transitions leave the respective split game position containing $i+$ and $i-$, too. Here, there are three existing outgoing game transitions, thus six satisfiability checks. For the first outgoing game transition one has $\mathrm{SAT}((n = 0) \wedge \mathrm{pre}(e)) = \mathrm{SAT}((n = 0) \wedge (n > 0)) = \mathit{false}$ and $\mathrm{SAT}((n > 0) \wedge \mathrm{pre}(e)) = \mathrm{SAT}((n > 0) \wedge (n > 0)) = \mathit{true}$, thus only one new weak game transition is added. Similar with the game transition leading to $\binom{\neg e}{\Box}$. The last existing outgoing weak game transition is somewhat special, as it is an incoming weak game transition, too. The two satisfiability checks are $\mathrm{SAT}((n = 0) \wedge \mathrm{pre}(tt)) = \mathrm{SAT}((n = 0) \wedge (n > 0)) = \mathit{false}$ and $\mathrm{SAT}((n > 0) \wedge \mathrm{pre}(tt)) = \mathrm{SAT}((n > 0) \wedge (n > 0)) = \mathit{true}$, thus again only one new weak game transition from $\binom{n>0}{\Diamond_2}$ to $\binom{tt}{\tilde{\wedge}}$ is created. This newly created game transition is an incoming weak game transition, too, and thus has to be checked. Furthermore it is the sole incoming weak game transition (the old existing one has been removed by the algorithm). The algorithm performs two more satisfiability

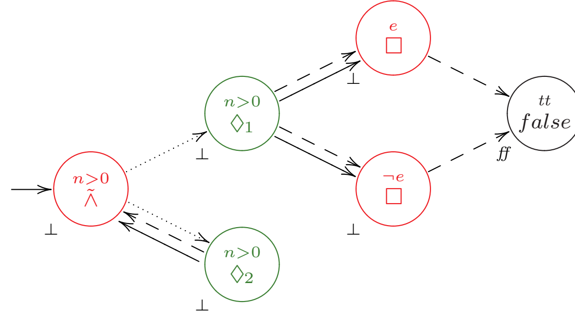

Figure 4.5: The *third turn: Simplification*.

checks and adds two new weak game transitions. The result can be seen in Figure 4.4 on the previous page.

The next step is checking every outgoing weak game transition for possible outgoing strong game transitions. There are four such game transitions, thus four unsatisfiability checks are performed. All four checked conditions are indeed unsatisfiable and four strong game transitions are added. Since there are no incoming strong game transitions, the algorithm `Split` finishes with the removal of the old game positions ($\in C'$).

A validity play results in no new information, but an invalidity play can identify three invalid game positions: $\binom{n=0}{\tilde{\wedge}}$, $\binom{n=0}{\Diamond_1}$, and $\binom{n=0}{\Diamond_2}$.

The resulting strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.4 on the preceding page.

Since the initial game position is still undecided, the third turn of the algorithm begins. This time the algorithm can indeed simplify the game. All game positions leaving $\binom{n=0}{\tilde{\wedge}}$ can be removed. Additionally all game transitions from $\binom{n>0}{\Diamond_2}$ to $\binom{n=0}{\tilde{\wedge}}$ can be removed, too. Finally the now unreachable game positions $\binom{n=0}{\tilde{\wedge}}$, $\binom{n=0}{\Diamond_1}$, and $\binom{n=0}{\Diamond_2}$ can be removed. The resulting simplified strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.5.

This time the *heuristic* selects a split of the game position $\binom{e}{\Box}$. The predicate selected by the heuristic is $(n = 0) \in \overline{\mathcal{L}}$.

Figure 4.6: The *third turn: Refinement*.

Similar to the previous splits, the algorithm `Split` determines the two new abstract worlds $i+$ and $i-$ with $\alpha(i+) = (n = 0) \wedge e$ and $\alpha(i-) = (n > 0) \wedge e$. Furthermore the algorithm determines the set C', adds two new game positions and adapts all game transitions. The resulting strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.6.

The only interesting difference is the calculation of the incoming strong game transitions. There is a single one leading from $\binom{n>0}{\Diamond_1}$ to $\binom{e}{\Box}$. Two unsatisfiability checks are performed:

$$\neg\mathrm{SAT}((n>0)\wedge\neg\mathrm{pre}(e\wedge(n=0)))=\neg\mathrm{SAT}((n>0)\wedge\neg(e\wedge(n>0)))=\neg\mathrm{SAT}((n>0)\wedge\neg e)=false$$

$$\neg\mathrm{SAT}((n>0)\wedge\neg\mathrm{pre}(e\wedge(n>0)))=\neg\mathrm{SAT}((n>0)\wedge\neg(\neg e\wedge(n>0)))=\neg\mathrm{SAT}((n>0)\wedge e)=false$$

Thus no incoming strong game transition can be added, i.e. the algorithm lost information. Fortunately this is detected by `Split` and the heuristic is informed to handle the game position $\binom{n>0}{\Diamond_1}$ next.

A validity play results in no new information, but a invalidity play can identify $\binom{(n>0)\wedge e}{\Box}$ as invalid. Thus all game transitions leaving that game position can be removed, in addition to the weak game transition entering the game position from the green C_0 game position. Finally the now unreachable game position $\binom{(n>0)\wedge e}{\Box}$ can be removed. The resulting simplified strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.7 on the next page.

This time the *heuristic* was informed of a possible next refinement, and thus selects the transformation of the game position $\binom{n>0}{\Diamond_1}$.

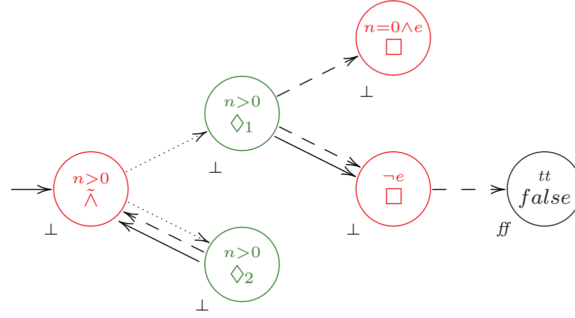

Figure 4.7: The *fourth turn: Simplification.*

The first step of the algorithm `Transform` is the calculation of the via junction game transitions connected set of game positions C'. This set consists of the three game positions $\binom{n>0}{\tilde{\wedge}}$, $\binom{n>0}{\Diamond_1}$, and $\binom{n>0}{\Diamond_2}$. Each of these game positions is transformed into a $C_{0/1}$ game position.

The next step is the calculation of the abstract BRANCH-worlds. The algorithm has to consider all possible subsets of I, but in this case there are only two subsets that pass the satisfiability check:

$$B_1 = (\{(tt),(n=0),(n>0),(e),(\neg e),((n=0)\wedge e)\},(n>0)\wedge e)$$
$$B_2 = (\{(tt),(n=0),(n>0),(e),(\neg e),((n>0)\wedge e)\},(n>0)\wedge \neg e)$$

For each game position in the set C' there is a new game position added for each of these BRANCH-worlds. Thus six new game positions are added in this example. The adaption of the junction game transitions as well as the connection of the OR game positions with the newly created BRANCH game positions is straightforward and can be seen in Figure 4.8 on the following page.

Additionally outgoing weak and strong game transitions have to be recalculated. Since whenever there is a weak game transition leaving a BRANCH game position, there will also be a strong one, the algorithm simply checks all outgoing weak game positions with the reachable abstract worlds encoded in the first component of each abstract BRANCH-world.

The complete refined strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.8 on the next page.

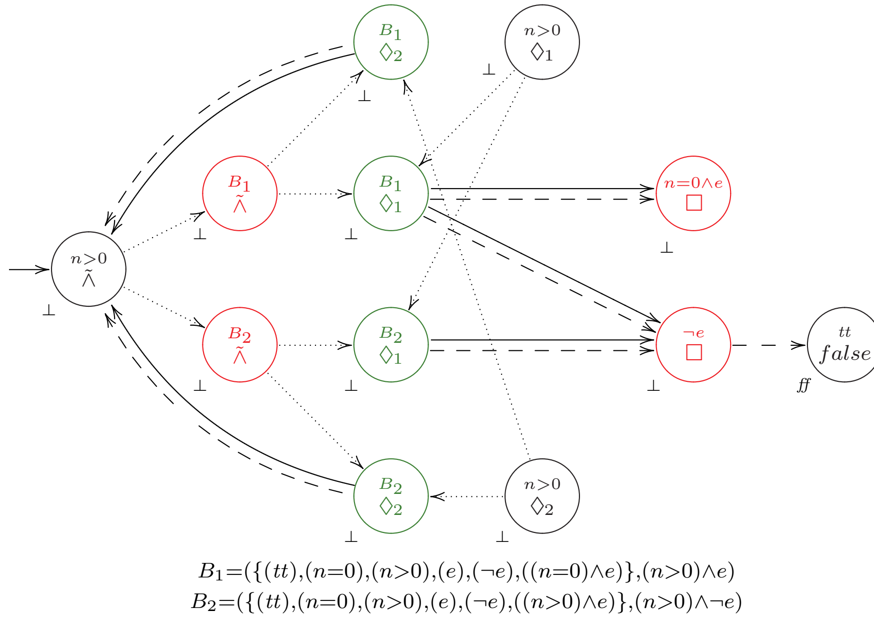

Figure 4.8: The *fourth turn: Transformation.*

A validity play identifies $\binom{B_1}{\Diamond_1}$ and $\binom{(n=0)\wedge e}{\Box}$ as valid game positions. An invalidity play results in no new information. Nevertheless the strong-weak-parity-game can be simplified. The result can be seen in Figure 4.9.

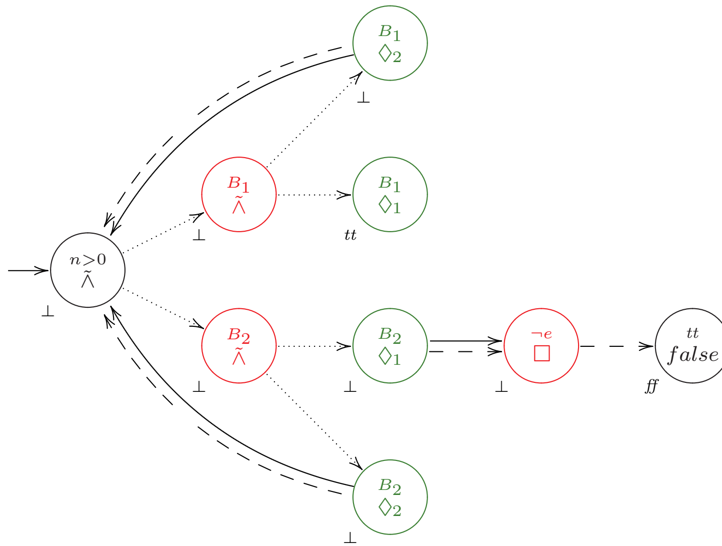

Figure 4.9: The *fifth turn: Simplification.*

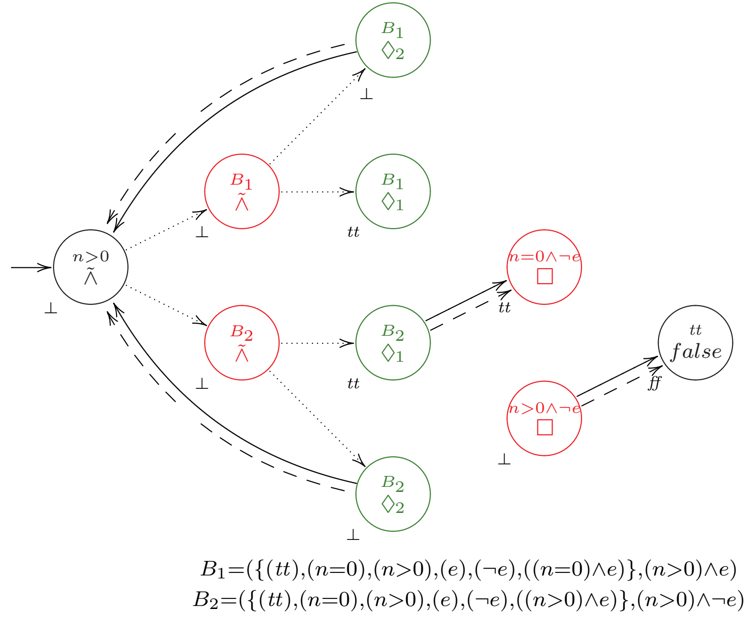

Figure 4.10: The *fifth turn: Refinement.*

In this fifth and last turn the heuristic picks the game position $\binom{\neg e}{\Box}$ for a split with the predicate $(n = o) \in \overline{\mathcal{L}}$.

The resulting strong-weak-parity-game (as part of the abstract property game) can be seen in Figure 4.10. A validity play identifies the game positions $\binom{(n=0\wedge\neg e}{\Box}$ and $\binom{B_2}{\Diamond_1}$ as valid. Now every validity play either ends at a valid game positions or is infinite, thus the initial game position is valid, too.

The algorithm stops and returns the validity of $\tilde{K}$ with respect to $\widetilde{A}$.

Chapter 5

Conclusion

We developed a new CEGAR-based algorithm for μ-calculus verification. Contrary to most other CEGAR-based algorithms, this algorithm operates directly upon an abstract property game. Thus there is no need to reconstruct the property game in each turn of the CEGAR-cycle. Furthermore each step of refinement is based upon the lazy abstraction technique that is applied locally in the game, i.e. only some game positions are refined instead of all game position belonging to an abstract world.

The algorithm begins with a given initial abstraction, which is based upon a most abstracted automaton. Than at each turn a heuristic determines the next step of the automatic abstraction refinement. This is continued until the given property can either be proved or disproved.

Furthermore this approach allows for a simplification of the abstract property game after each refinement step, thus further reducing the state explosion problem.

5.1 Future Work

A more detailed discussion of possible heuristics, especially additional heuristics that imply the use of `Transform`, is a topic for future work.

Furthermore we gave no notion of completeness. The algorithm should be complete for least fixpoint free μ-calculus formulas, i.e. in the alternating tree automata, as well as in the abstract property game, the acceptance condition respective the parity function always maps to zero. A detailed analysis and a proof similar to the proof of soundness in this work remains for future work.

Finally an implementation of the algorithm, similar to the implementation of the algorithm of Fecher and Shoham [8] by Fenten [9] in his (as yet unpublished) diploma thesis, is a topic for future work.

Algorithms

Algorithm `PropertyCheck`(K, A)

1 Set the abstract property game $P_{K,A}$ to $P^0_{K,A}$
2 `while` $(\omega(\hat{c}) = \bot)$ `do`
3 `Simplify`(G)
4 Remove from G every game position $c \in C$ that is unreachable from $\hat{c}$ and every game transition leaving such a game position c.
5 `Refine`$(P_{K,A})$
6 `return` $\omega(\hat{c})$

Algorithm `Simplify`(G)

1 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
2 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
3 Remove all game transitions (c, c') from G with $\omega(c) \neq \bot$.
4 Remove all game transitions (c, c') from G with $c \in C_0$ and $\omega(c') = ff$.
5 Remove all game transitions (c, c') from G with $c \in C_1$ and $\omega(c') = tt$.
6 `Return` the modified game G as G'.

Algorithm `Refine`$(P_{K,A})$

1 A Heuristic determines the used refinement approach.
2 *% Heuristic determines split of game position c with predicate p with $c = (i, q)$ and $i \in I$*
3 `Split`$(P_{K,A}$,c,$p)$
4 *% Heuristic determines transformation of game position c with $c = (i, q)$, $i \in I$, and $\iota(i) = \gamma$*
5 `Transform`$(P_{K,A}$,$c)$
6 `return` the modified game $P_{K,A}$

Algorithm `OptimizedPropertyCheck`(K, A)

1 Set the abstract property game $P_{K,A}$ to $P^0_{K,A}$
2 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
3 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
4 **while** $(\omega(\hat{c}) = \bot)$ **do**
5 Remove all game transitions (c, c') from G with $\omega(c) \neq \bot$.
6 Remove all game transitions (c, c') from G with $c \in C_0$ and $\omega(c') = ff$.
7 Remove all game transitions (c, c') from G with $c \in C_1$ and $\omega(c') = tt$.
8 Remove from G every game position $c \in C$ that is unreachable from $\hat{c}$ and every game transition leaving such a game position c.
9 A Heuristic determines the used refinement approach.
10 *% Heuristic determines split of game position c with predicate p with* $c = (i, q)$ *and* $i \in I$
11 `Split`$(P_{K,A}, c, p)$
12 *% Heuristic determines transformation of game position c with* $c = (i, q)$, $i \in I$, *and* $\iota(i) = \gamma$
13 `Transform`$(P_{K,A}, c)$
14 Use a parity game algorithm to determine the valid game positions and set $\omega(c) = tt$ in G for every valid game position c.
15 Use a parity game algorithm to determine the invalid game positions and set $\omega(c) = ff$ in G for every invalid game position c.
16 **return** $\omega(\hat{c})$

Procedure `Add`$(P_{K,A}, (i, q), \vartheta', \omega')$

1 **if** $i \in I \wedge \iota(i) = \mu$ *% OR−world*
2 $C_{0/1} := C_{0/1} \cup (i, q)$
3 **else** *% TRANS−world or BRANCH−world*
4 **if** $q \in Q_0$ **then**
5 $C_0 := C_0 \cup (i, q)$
6 **else if** $q \in Q_1$ **then**
7 $C_1 := C_1 \cup (i, q)$
8 **else** *%* $q \in Q_{0/1}$
9 $C_{0/1} := C_{0/1} \cup (i, q)$
10 **if** $\alpha(i) \Rightarrow \delta(q)$
11 $\omega' := tt$
12 **else if** $\alpha(i) \Rightarrow \neg\delta(q)$
13 $\omega' := ff$
14 **else**
15 $\omega' := \bot$
16 *% Adapt* ϑ *and* ω
17 Set $\vartheta((i, q)) := \vartheta'$
18 Set $\omega((i, q)) := \omega'$

Algorithm `Split`($P_{K,A}$,c,p)

1 *% Splitting the abstract world*
2 determine $[\![\alpha(i) \wedge p]\!]$ and $[\![\alpha(i) \wedge \neg p]\!]$
3 **add** an $i+$ to I, adapt α so that $\alpha(i+) = (\alpha(i) \wedge p)$, and adapt ι so that $\iota(i+) = \iota(i)$.
4 **add** an $i-$ to I, adapt α so that $\alpha(i-) = (\alpha(i) \wedge \neg p)$, and adapt ι so that $\iota(i-) = \iota(i)$.
5 *% Calculate the set of game positions C'*
6 $C? := \{(i,q)\}$; $C' := \emptyset$
7 **while** $C? \neq \emptyset$ **do**
8 remove (i',q') from $C?$
9 $C' := C' \cup \{(i',q')\}$
10 **foreach** $(((i?,q?),(i',q')) \in G^\circ$ **or** $((i',q'),(i?,q?)) \in G^\circ)$ **do**
11 $C? := C? \cup \{(i?,q?)\} \setminus C'$
12 *% Add each new game position and adjust the initial game position*
13 **foreach** $(i,q') \in C'$ **do**
14 `Add`($P_{K,A}$, $(i+,q')$,$\vartheta((i,q'))$, $\omega((i,q'))$)
15 `Add`($P_{K,A}$, $(i-,q')$,$\vartheta((i,q'))$, $\omega((i,q'))$)
16 **if** $\hat{c} = (i,q')$ **then**
17 **if** `Satisfiable`($\hat{p} \wedge p$) **then** $\hat{c} := (i+,q')$ **else** $\hat{c} := (i-,q')$
18 *% Split in the pre−abstraction part.*
19 **if** $\iota(i) = \gamma$ **then**
20 *% Add new junction game transitions*
21 **foreach** $(i,q') \in C'$ **do**
22 **foreach** $((i,q'),(i,q?)) \in G^\circ$ **do**
23 remove $((i,q'),(i,q?))$ from G°
24 insert $((i+,q'),(i+,q?))$ into G°
25 insert $((i-,q'),(i-,q?))$ into G°
26 *% Add new outgoing weak game transitions*
27 **foreach** $(i,q') \in C'$ **do**
28 **foreach** $((i,q'),(i?,q?)) \in G^+$ **do**
29 remove $((i,q'),(i?,q?))$ from G^+
30 **if** `Satisfiable`($\alpha(i+) \wedge \mathrm{pre}(\alpha(i?))$) **then** insert $((i+,q'),(i?,q?))$ into G^+
31 **if** `Satisfiable`($\alpha(i-) \wedge \mathrm{pre}(\alpha(i?))$) **then** insert $((i-,q'),(i?,q?))$ into G^+
32 *% Add new incoming weak game transitions*
33 **foreach** $(i,q') \in C'$ **do**
34 **foreach** $((i?,q?),(i,q')) \in G^+$ **do**
35 remove $((i?,q?),(i,q'))$ from G^+
36 **if** `Satisfiable`($\alpha(i?) \wedge \mathrm{pre}(\alpha(i+))$) **then** insert $((i?,q?),(i+,q'))$ into G^+
37 **if** `Satisfiable`($\alpha(i?) \wedge \mathrm{pre}(\alpha(i-))$) **then** insert $((i?,q?),(i-,q'))$ into G^+
38 *% Add new outgoing strong game transitions*
39 **foreach** $(i,q') \in C'$ **do**
40 **foreach** $((i+,q'),(i?,q?)) \in G^+$ **do**
41 **if** $((i,q'),(i?,q?)) \in G^-$ **then**
42 insert $((i+,q'),(i?,q?))$ into G^-
43 **else**
44 **if not** `Satisfiable`($\alpha(i+) \wedge \neg\mathrm{pre}(\alpha(i?))$) **then** insert $((i+,q'),(i?,q?))$ into G^-
45 **foreach** $((i-,q'),(i?,q?)) \in G^+$ **do**
46 **if** $((i,q'),(i?,q?)) \in G^-$ **then**
47 insert $((i-,q'),(i?,q?))$ into G^-
48 **else**
49 **if not** `Satisfiable`($\alpha(i-) \wedge \neg\mathrm{pre}(\alpha(i?))$) **then** insert $((i-,q'),(i?,q?))$ into G^-
50 **foreach** $((i,q'),(i?,q?)) \in G^-$ **do**
51 remove $((i,q'),(i?,q?))$ from G^-

```
52 % Add new incoming strong game transitions
53     foreach (i, q') ∈ C' do
54       foreach ((i?, q?), (i, q')) ∈ G⁻ do
55         remove ((i?, q?), (i, q')) from G⁻
56         added := false;
57         if ((i?, q?), (i+, q')) ∈ G⁺ then
58           if not Satisfiable(α(i?) ∧ ¬pre(α(i+))) then
59             insert ((i?, q?), (i+, q')) into G⁻
60             added := true;
61         if ((i?, q?), (i−, q')) ∈ G⁺ then
62           if not Satisfiable(α(i?) ∧ ¬pre(α(i−))) then
63             insert ((i?, q?), (i−, q')) into G⁻
64             added := true;
65         if not added then
66           Notify Heuristic of game position (i?, q?).
67 % Split in the post−abstraction part.
68   if ι(i) = μ then
69 % Collect all BRANCH worlds in C'
70     B' := ∅
71     foreach ((B, P), q') ∈ C' do
72       B' := B' ∪ {B}
73 % Split each BRANCH world
74     foreach B ∈ B' do
75       foreach I' ∈ ℙ(B) do
76         ϕ+ := α(i+) ∧ ⋀_{i'∈I'} pre(α(i')) ∧ ⋀_{i'∈B\I'} ¬pre(α(i'))
77         ϕ− := α(i−) ∧ ⋀_{i'∈I'} pre(α(i')) ∧ ⋀_{i'∈B\I'} ¬pre(α(i'))
78         if Satisfiable(ϕ+) then B+ := (I', ϕ+)
79         if Satisfiable(ϕ−) then B− := (I', ϕ−)
80 % Add new BRANCH game positions
81     foreach ((B, P), q') ∈ C' do
82       Add(P_{K,A}, (B+, q'), ϑ(((B, P), q')), ω(((B, P), q')))
83       Add(P_{K,A}, (B−, q'), ϑ(((B, P), q')), ω(((B, P), q')))
84 % Add new OR to BRANCH junction game transitions
85     foreach (i, q') ∈ C' with ∈ I do
86       foreach ((i, q'), ((B, P), q')) ∈ G° do
87         remove ((i, q'), (B, P), q')) from G°
88         insert ((i+, q'), (B+, q')) into G°
89         insert ((i−, q'), (B−, q')) into G°
90 % Add new BRANCH to BRANCH junction game transitions
91     foreach ((B, P), q') ∈ C' do
92       foreach (((B, P), q'), ((B, P), q?)) ∈ G° do
93         remove (((B, P), q'), (B, P), q?)) from G°
94         insert ((B+, q'), (B+, q?)) into G°
95         insert ((B−, q'), (B−, q?)) into G°
96 % Add new outgoing weak and strong game transitions
97     foreach ((B, P), q') ∈ C' do
98       foreach (((B, P), q'), (i?, q?)) ∈ G⁺ do
99         remove (((B, P), q'), (i?, q?)) from G⁺
100        remove (((B, P), q'), (i?, q?)) from G⁻
101        if i? ∈ Π₁(B+) then
102          insert ((B+, q'), (i?, q?)) into G⁺
103          insert ((B+, q'), (i?, q?)) into G⁻
104        if i? ∈ Π₁(B−) then
```

```
105             insert  ((B−, q'), (i?, q?)) into G+
106             insert  ((B−, q'), (i?, q?)) into G−
107 % Add new incoming weak game transitions
108     foreach (i, q') ∈ C' with i ∈ I do
109       foreach ((i?, q?), (i, q')) ∈ G+ do
110         remove ((i?, q?), (i, q')) from G+
111         if Satisfiable(α(i?) ∧ pre(α(i+))) then insert ((i?, q?), (i+, q')) into G+
112         if Satisfiable(α(i?) ∧ pre(α(i−))) then insert ((i?, q?), (i−, q')) into G+
113 % Add new incoming strong game transitions
114     foreach (i, q') ∈ C' with i ∈ I do
115       foreach ((i?, q?), (i, q')) ∈ G− do
116         remove ((i?, q?), (i, q')) from G−
117         added := false;
118         if ((i?, q?), (i+, q')) ∈ G+ then
119           if not Satisfiable(α(i?) ∧ ¬pre(α(i+))) then
120             insert  ((i?, q?), (i+, q')) into G−
121             added := true;
122         if ((i?, q?), (i−, q')) ∈ G+ then
123           if not Satisfiable(α(i?) ∧ ¬pre(α(i−))) then
124             insert  ((i?, q?), (i−, q')) into G−
125             added := true;
126         if not added then
127           Notify Heuristic of game position (i?, q?).
128   % Add new incoming weak game transitions
129     foreach (i, q') ∈ C' do
130       remove (i, q') from P_{K,A} and adjust ϑ and γ
```

Algorithm `Transform`($P_{K,A}$,c)

1 *% Calculate the set of game positions* C'
2 $C? := \{(i, q)\}$; $C' := \emptyset$
3 `while` $C? \neq \emptyset$ `do`
4 remove (i, q') from $C?$
5 $C' := C' \cup \{(i, q')\}$
6 `foreach` $(((i?, q?), (i, q')) \in G^\circ$ `or` $((i, q'), (i?, q?)) \in G^\circ)$ `do`
7 $C? := C? \cup \{(i?, q?)\} \setminus C'$
8 *% Transform all TRANS game positions into OR game positions*
9 $\iota(i) := \mu$
10 `foreach` $c' \in C'$
11 Move c' into $C_{0/1}$
12 *% Calculate BRANCH worlds*
13 $B' := \emptyset$
14 `foreach` $I' \in \mathbb{P}(I)$ `do`
15 $\phi := \alpha(i) \wedge \bigwedge_{i' \in I'} \text{pre}(\alpha(i')) \wedge \bigwedge_{i' \in I \setminus I'} \neg\text{pre}(\alpha(i'))$
16 `if Satisfiable`(ϕ) `then`
17 $B' := B' \cup \{(I', \phi)\}$
18 *% Add BRANCH game positions*
19 `foreach` $(i, q') \in C'$ `do`
20 `foreach` $B \in B'$ `do`
21 `Add`($P_{K,A}$,(B, q'),$\vartheta((i, q'))$,$\omega((i, q'))$)
22 *% Adapt junction game transitions*
23 `foreach` $(i, q') \in C'$ `do`
24 `foreach` $((i, q'), (i, q'')) \in G^\circ$
25 remove $((i, q'), (i, q''))$ from G°
26 `foreach` $B \in B'$ `do`
27 insert $((B, q'), (B, q''))$ into G°
28 *% Connect OR to BRANCH game positions*
29 `foreach` $(i, q') \in C'$ `do`
30 `foreach` $B \in B'$ `do`
31 insert $((i, q'), (B, q'))$ into G°
32 *% Adapt outgoing weak and strong game transitions*
33 `foreach` $(i, q') \in C'$ `do`
34 `foreach` $((i, q'), (i?, q?)) \in G^+$
35 remove $((i, q'), (i?, q?))$ from G^+
36 remove $((i, q'), (i?, q?))$ from G^-
37 `foreach` $B \in B'$ `do`
38 `if` $i? \in \Pi_1(B)$ `then`
39 insert $((B, q'), (i?, q?))$ into G^+
40 insert $((B, q'), (i?, q?))$ into G^-

List of Figures

2.1 A finite state machine. 5
2.2 A pointed Kripke structure. 7
2.3 Transitions in alternating tree automata. 9
2.4 An alternating tree automaton. 10
2.5 A property game. 15

3.1 A generalized Kripke modal transition system. 18
3.2 A μ-automaton. 19
3.3 A generalized μ-automaton. 22
3.4 A property game. 27
3.5 The initial generalized μ-automaton. 28
3.6 The initial property game. 29

4.1 The CEGAR-Cycle. 37
4.2 The initial abstract property game. 57
4.3 The first turn: Refinement. 58
4.4 The second turn: Refinement. 59
4.5 The third turn: Simplification. 60
4.6 The third turn: Refinement. 61
4.7 The fourth turn: Simplification. 62
4.8 The fourth turn: Transformation. 63
4.9 The fifth turn: Simplification. 63
4.10 The fifth turn: Refinement. 64

Bibliography

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[2] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[3] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[4] Luca de Alfaro, Patrice Godefroid, and Radha Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *LICS*, pages 170–179, 2004.

[5] E. Allen Emerson. Model checking and the μ-calculus. In *Descriptive Complexity and Finite Models*, pages 185–214, 1996.

[6] Harald Fecher and Michael Huth. More precise partition abstraction. In *VMCAI*, Lecture Notes in Computer Science. Springer, 2007.

[7] Harald Fecher and Sharon Shoham. Local abstraction-refinement for the μ-calculus. In *SPIN*, volume 4595 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Harald Fecher and Sharon Shoham. State focusing: Lazy abstraction for the μ-calculus. In *SPIN*, volume 5156 of *Lecture Notes in Computer Science*. Springer, 2008.

[9] Tim Fenten. A model checker for the modal mu-calculus using lazy abstraction refinement. Dimploma Thesis, Christian-Albrechts-University Kiel, Germany, 2008.

[10] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[11] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. Don't know in the μ-calculus. In *VMCAI*, pages 233–249, 2005.

[12] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Inf. Comput.*, 205(8):1130–1148, 2007.

[13] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *TACAS*, pages 212–226, 2006.

[14] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[15] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27: 333–354, 1983.

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

[17] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Symposium on Programming*, pages 337–351, 1982.

[18] Jan Waller. Generalized μ-automata, satisfaction and refinement games. Student Research Project, Christian-Albrechts-University Kiel, Germany, November 2007.

[19] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Belg. Math. Soc.*, 8(2):359–391, May 2001.