Abstraction-Refinement and Modularity in μ-Calculus Model Checking

Sharon Shoham

Abstraction-Refinement and Modularity in μ-Calculus Model Checking

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Sharon Shoham

Submitted to the Senate of the
Technion - Israel Institute of Technology

Shvat 5769 Haifa February 2009

This research thesis was done under the supervision of Prof. Orna Grumberg in the Department of Computer Science.

I would like to thank Orna, my advisor, who has assisted me and guided me throughout the long years of my work. For many long academic, as well as social, conversations, and most of all for your great support and encouragement. I highly appreciate the good fortune of having the chance to work with you.

I would like to thank the IBM T.J. Watson research lab at Hawthorne for the opportunity of doing a summer internship there and for the generous support during my visit. Special thanks goes to the researchers from this lab for their collaborations on interesting research topics.

I would like to thank my many friends at the Technion, and outside, which have been a constant source of support and help. For many interesting coffee breaks at the 3rd floor that enriched my world.

I would like to thank my parents, Carmela and Doron, who were always there for me, assisting me. To my beloved sisters, Keren and Chen, who were my link to the Technion whenever I was away, running annoying errands with no complaint.

Finally, I would like to thank my husband, Niv, for convincing me to continue to graduate studies. For his support and love all of these years, through the pressure, the travels, and the deadlines. I couldn't have made it without you!

The generous financial help of the Technion is gratefully acknowledged.

Contents

Abstract **1**

Notation and Abbreviations **3**

1 Introduction **5**
- 1.1 Abstraction-Refinement 5
- 1.2 Compositional Verification 7
- 1.3 Overview of the Thesis 7

2 Preliminaries **11**
- 2.1 The μ-calculus 11
- 2.2 Concrete Models and Concrete Semantics 12
- 2.3 Abstraction and 3-Valued Semantics 13

3 3-Valued Model Checking Games **19**
- 3.1 Introduction 19
  - 3.1.1 Related work 20
- 3.2 Model Checking Games for 3-Valued μ-Calculus 20
- 3.3 3-Valued Parity Games 27
  - 3.3.1 3-Valued Parity Games 27
  - 3.3.2 Model Checking Games as Parity Games 28
- 3.4 Concluding Remarks 30

4 Game-Based Abstraction-Refinement: Direct Approach **31**
- 4.1 Introduction 31
  - 4.1.1 Related Work 32
- 4.2 Model Checking via Solving 3-Valued Parity Games 33
  - 4.2.1 Solving 3-Valued Parity Games 34
- 4.3 Refinement of 3-Valued Parity Games 45
  - 4.3.1 Identifying a Failure Cause 45
  - 4.3.2 Failure Analysis 51
- 4.4 Incremental Abstraction-Refinement Framework 52
- 4.5 Concluding Remarks 53

5 Game-Based Abstraction-Refinement: Reduction Approach 55
5.1 Introduction 55
5.1.1 Related Work 56
5.2 Solving 3-Valued Parity Games via Reduction 57
5.3 Refinement 58
5.3.1 Using Non-Losing Strategies to Solve the Game 59
5.3.2 Refinement with Non-Losing Strategies 60
5.4 Concluding Remarks 63

6 Monotonic Abstraction-Refinement 65
6.1 Introduction 65
6.1.1 Related Work 66
6.2 Generalized Abstract Models 67
6.2.1 Motivation 67
6.2.2 Generalized KMTSs 69
6.3 Monotonic Abstraction-Refinement Framework 73
6.4 Concluding Remarks 78

7 More Precision at Less Cost 79
7.1 Introduction 79
7.1.1 Related Work 82
7.2 Abstraction Framework 84
7.3 Increasing Precision 86
7.3.1 Precision of Abstract Models 87
7.3.2 May Transitions as a Source of Imprecision 91
7.3.3 Hyper Kripke Modal Transition Systems 91
7.4 Decreasing the Model Checking Cost 95
7.4.1 Optimized Abstract Model Checking 98
7.5 Abstraction-Refinement 103
7.6 Handling Multiple Initial States 104
7.7 Concluding Remarks 106

8 Compositional Verification and 3-Valued Abstractions Join Forces 107
8.1 Introduction 107
8.1.1 Related Work 109
8.2 Preliminaries 111
8.2.1 Game Graph 111
8.2.2 Coloring Algorithm 111
8.3 Partial Coloring and Subgraphs 113
8.4 Compositional Model Checking 115
8.5 Adding Abstraction 120
8.5.1 Motivation 121
8.5.2 Compositional Abstraction-Refinement 123
8.6 Extension: Labeled Transition Systems 131
8.7 Concluding Remarks 133

9 Conclusion **135**

Bibliography **137**

List of Figures

2.1 (a) A concrete Kripke structure, and (b) an abstract KMTS for it. 17

3.1 The model checking game rules for 3-valued μ-calculus. 21
3.2 All possible moves in the game over the KMTS from Figure 2.1(b). 23
3.3 The parity game corresponding to the game from Figure 3.2. 29

4.1 Illustration of the sets used by `ComputeOpponentWin`, and of those used by `ComputeNoWin`, in which the must-attractor is replaced by a may-attractor and vice versa. 36
4.2 Computation of winning vertices for the opponent: `ComputeOpponentWin`. 36
4.3 Computation of vertices in which no win is possible: `ComputeNoWin`. . . . 39
4.4 Algorithm `SolveGame`. 41
4.5 Subgame $\mathcal{G}[Y]$ used by `SolveGame` ($\mathcal{G}$) in Example 4.14. 44
4.6 Illustration of the failure analysis for the 3-valued parity game $\mathcal{G}$ depicted in Figure 3.3. 51

5.1 The reduced games of the parity game from Figure 3.3. 58
5.2 Winning strategies of the reduced games of the parity game from Figure 3.3. 61

6.1 (a) An abstract model M describing the program P, and (b) the abstract model M' resulting from its refinement. Outgoing transitions of s_{21} are omitted since they are irrelevant. 68
6.2 The model M'' achieved by applying refinement as suggested in [37] on M from Figure 6.1(a). Outgoing transitions of s_{21} are omitted since they are irrelevant, and so are additional outgoing may transitions of the unrefined states (there are no additional outgoing must transitions for the unrefined states). 69
6.3 New model checking game rules for GTSs. 77

7.1 Illustration of Example 7.1. 80
7.2 Rules for the construction of the model checking graph. 96
7.3 A colored model checking graph. Gray vertices are colored ?, while white vertices are colored T. 103

8.1 Compositional model checking algorithm. 120

8.2 (a) Components, (b) their game graphs and their ?-subgraphs (enclosed by a line), and (c) the product graph. Dashed edges denote may edges which are not must edges. The colors reflect the coloring function: white stands for T, dark gray stands for F and light gray stands for ?. 121
8.3 Compositional abstraction-refinement algorithm. 127
8.4 Most coarse abstraction of M_2 from Figure 8.2(a) w.r.t. $\{r\}$. 128
8.5 Game graphs arising during the run of the compositional abstraction-refinement algorithm described in Example 8.28. 129
8.6 (a) Pruned ?-subgraph, and (b) its re-coloring after refinement which changes the color of $\hat{t}_0 \vdash \neg i$ to T. 131

Abstract

Model checking is an automated technique for checking whether or not a given system model fulfills a desired property, described as a temporal logic formula. Yet, as real models tend to be very big, model checking encounters the state-explosion problem, which refers to its high space requirements. Two of the most successful approaches to fighting the state-explosion problem in model checking are abstraction and compositional model checking.

Abstractions hide certain details of the system in order to result in a smaller model. In some cases the abstraction is too coarse, resulting in an inconclusive model checking result. Thus, the abstract model is *refined* by adding more details into it, making it more similar to the concrete model. Iterating this process is called *abstraction-refinement*. In *compositional* model checking, on the other hand, one tries to verify parts of the system separately in order to avoid the construction of the entire system.

In this research, we investigate abstraction-refinement and compositional techniques for specifications in the μ-calculus, which is a powerful formalism for expressing properties of transition systems using fixpoint operators. Our work exploits and extends the game-based approach to μ-calculus model checking, in which the model checking problem is formulated as a game between a verifier and a falsifier. We develop novel abstraction-refinement schemes for the μ-calculus, based on a 3-valued semantics. The 3-valued semantics allows an abstract model to be used for verification as well as falsification, unlike traditional abstraction which is only used for verification. We investigate both the efficiency and the precision of abstract models used within the abstraction-refinement framework. We then extend our work on abstraction-refinement to the arena of compositional verification, thus joining the forces of both approaches. We use techniques taken from the game-based 3-valued model checking for abstract models to obtain a novel fully automatic compositional technique that can determine the truth value of the full μ-calculus w.r.t. a given system.

Notation and Abbreviations

KMTS	— Kripke Modal Transition System
GTS	— Generalized Kripke Modal Transition System
HTS	— Hyper Kripke Modal Transition System
AP	— Set of atomic propositions
Lit	— Set of literals (atomic propositions and their negations)
$\mathcal{L}_\mu$	— The μ-calculus
$\mathcal{L}_\mu^0$	— The alternation-free fragment of the μ-calculus
φ	— μ-calculus formula
$Sub(\varphi)$	— Subformulas of φ
$fp(Z)$	— Fixpoint formula associated with variable Z
M	— Model of a system (Kripke structure, KMTS, GTS or HTS)
tt	— Truth value 'true'
ff	— Truth value 'false'
$\perp$	— Truth value 'indefinite'
ρ	— Environment, explaining the meaning of free variables in a formula
$[\![\varphi]\!]^{M,\rho}$	— Concrete semantics of φ w.r.t. a Kripke structure M and an environment ρ
$[\![\varphi]\!]_3^{M,\rho}$	— 3-Valued semantics of φ w.r.t. a KMTS, GTS or HTS M and an environment ρ
$M \models \varphi$	— The model M satisfies φ
$M \not\models \varphi$	— The model M falsifies φ
$M \overset{?}{\models} \varphi$	— The value of φ in M is indefinite
$\preceq$	— Mixed, Generalized mixed, or Hyper mixed simulation relation
S_C	— Set of concrete states
S_A	— Set of abstract states
γ	— Concretization function
M_C	— Concrete model (Kripke structure)
M_A	— Abstract model (KMTS, GTS or HTS)
$\Gamma_M(s, \varphi)$	— 3-valued model checking game on a model M, a state s and a formula φ

Chapter 1

Introduction

This research deals with abstraction-refinement and modularity in model checking, a key procedure in the area of formal verification. Model checking [19] is a useful approach for verifying properties of systems. It is given a model M of a system and a temporal logic formula φ, describing a specification, and returns 'true' if the system satisfies the specification and 'false', otherwise.

The logic specifications we consider in this research are formulas of the modal *μ-calculus* [50]. The μ-calculus is a powerful formalism for expressing properties of transition systems using fixpoint operators. Many verification procedures can be solved by translating them into μ-calculus model checking [13]. Such problems include (fair) CTL model checking, LTL model checking, bisimulation equivalence and language containment of deterministic ω-regular automata.

The main disadvantage of model checking in general, and in the context of μ-calculus properties in particular, is the *state explosion problem*, which refers to its high space requirements. Several solutions have been suggested in the literature to fight the state explosion problem. Two of the most promising approaches are abstraction and compositional verification, which are also the subject of this research.

1.1 Abstraction-Refinement

Abstractions hide certain details of the system in order to result in a smaller (abstract) model. Most commonly used are state abstractions that collapse (possibly non disjoint) sets of concrete states into abstract states. An abstract model is then constructed in a way that ensures preservation of the logic of interest from the abstract model to the concrete model.

Two types of semantics are available for interpreting temporal logic formulas over abstract models. The *2-valued* semantics defines a formula φ to be either *true* or *false* in an abstract model. When using a 2-valued semantics, abstract models are usually designed to be *conservative* for *true*, meaning that if a formula is true of the abstract model then it is also true of the concrete (precise) model of the system. However, if it is false in the abstract model then nothing can be deduced of the concrete one.

In order to obtain more precise results, temporal logics can be interpreted over abstract models w.r.t. the *3-valued* semantics [11, 45, 39]. The *3-valued* semantics evaluates a formula to either *true*, *false*, or *indefinite*. Abstract models can then be designed to be conservative for both *true* and *false*, meaning that both truth and falsity of a formula are preserved from the abstract model to the concrete model. Only if the value of a formula in the abstract model is indefinite, its value in the concrete model is unknown. Abstractions over 3-valued semantics thus give precise results more often both for verification and falsification.

In either setting, the abstraction is sometimes too coarse, resulting in an inconclusive model checking result. In this case, the abstract model is *refined* by adding more details into it, making it more similar to the concrete model. Refinement is traditionally done by splitting abstract states based on some criterion. Iterating this process of abstraction, model checking, and refinement is called *abstraction-refinement*.

The traditional abstraction-refinement framework [52, 18] is designed for universal temporal logics, such as the *linear time logic* LTL, and the universal fragments of the *branching time logics* CTL, CTL* and the μ-calculus [19]. It considers 2-valued abstractions, where false may be a false-alarm, thus refinement is aimed at eliminating false results. As such, it is usually based on a counterexample analysis. In case that the abstract model does not satisfy the property, it is known how to verify whether the found error occurs also in the concrete (full) model. If not, then the spurious error is used in order to refine the abstract model. This approach is less suitable for branching time temporal logics with both universal and existential operators, such as the full μ-calculus, where the notion of counterexample is less clear.

Moreover, the traditional abstraction framework considers either over-approximated abstract models (for verification of universal properties) or under-approximated abstract models (for their falsification). Abstractions for the full μ-calculus, on the other hand, require more complex abstract models: Two transition relations are needed in an abstract model for it to be conservative w.r.t. the full μ-calculus, be it over a 2-valued or a 3-valued semantics. Examples of such abstract models are *Mixed Transition Systems* [26] or *Kripke Modal Transition Systems* (KMTSs) [45, 38], which extend *Modal Transition Systems* [56, 54][1]. These models contain *may* transitions which over-approximate transitions of the concrete model, and *must* transitions, which under-approximate the concrete transitions. To ensure logic preservation, truth of universal formulas is then examined over may transitions, whereas truth of existential formulas is examined over must transitions. Dually for falsity when a 3-valued semantics is considered.

While the traditional abstraction-refinement has been investigated intensively, not much research has been devoted to abstraction-refinement algorithms for branching time temporal logics that combine both existential and universal quantifiers. Several works [65, 66, 60, 6] suggested abstraction-refinement mechanisms for the μ-calculus and CTL over *2-valued* semantics, for *specific* abstractions. The refinement in these works is abstraction-specific. It is not suitable for every abstraction. As for the 3-valued semantics, several researchers studied model checking for abstract models w.r.t. specifications in μ-calculus, interpreted over a 3-valued semantics (e.g. [37, 39]). Yet, these works lack an automatic refinement mechanism, which prevents them from comprising an automatic abstraction-refinement framework. No general, automatic abstraction-refinement framework based on a 3-valued semantics for the μ-calculus was suggested in the literature.

[1]In this research we use KMTSs as a starting point for our investigation of abstract models.

1.2 Compositional Verification

Another promising solution to the state explosion problem is *compositional* model checking, where parts of the system are verified separately in order to avoid the construction of the entire system and reduce the model checking cost. Usually, it is impossible to verify a component of the system in complete isolation from the rest of the system. This is because the behavior of one component depends on the interaction (e.g., input-output) it has with its environment.

To account for the dependencies between the components, the Assume-Guarantee (AG) paradigm [46, 68] suggests how to verify one component based on an *assumption* about the behavior of its environment, where the environment consists of the other system components. The environment is then verified, in order to *guarantee* that it actually satisfies the assumption.

Many of the works on compositional model checking are based on the AG paradigm. Various AG proof rules have been suggested in the literature. A common obstacle in the application of all of them is the need to construct assumptions which are on the one hand simple enough to enable efficient verification, and on the other hand detailed enough to capture the properties of the environment that ensure correct behavior of the component. In practice, the development of the assumptions for each component becomes a bottleneck in modular verification. This is because it requires deep knowledge of the design and in many cases the first version is either incorrect or too abstract.

Most of the works on AG reasoning do not tackle the problem of automatically constructing assumptions and checking their correctness. The latter is particularly problematic since the environment, being the rest of the system, is typically very large. To overcome this difficulty, recent works suggested automatic assumption generation based on *learning* [23, 5, 15, 34]. They use iterative AG reasoning, where in each iteration the assumptions are modified based on the learning algorithm. These works are all designed for universal properties, mostly (except for [34]) safety properties. Their extension to full branching time logics such as the μ-calculus is unclear.

1.3 Overview of the Thesis

This thesis develops novel abstraction-refinement schemes, as well as compositional model checking techniques, for the verification of μ-calculus specifications. In what follows we give an overview of our results.

Game-Based 3-Valued Abstraction-Refinement This work presents a novel game-based approach to abstraction-refinement for the full μ-calculus, interpreted over 3-valued semantics.

We define a new game for the 3-valued model checking problem of the μ-calculus. Similarly to the traditional game-based approach to model checking [75], the game is defined s.t. the player that has a winning strategy in the game determines the model checking result. To account for the indefinite truth value, it is possible that none of the players has a winning strategy. Model checking then reduces to the problem of solving the model checking game, namely determining the player that has a winning strategy. The game is described in Chapter 3.

We derive from the game two abstraction-refinement schemes. Each scheme consists of a game-based model checking algorithm for abstract models w.r.t. specifications in μ-calculus interpreted over a 3-valued semantics, as well as an automatic refinement mechanism. The 3-valued semantics allows the model checking to be used for verification as well as falsification. Refinement in this case is aimed at eliminating indefinite results of the model checking, rather than false results. Namely, if the model checking result is indefinite, the abstract model is refined, based on an analysis of the cause for this result.

The first abstraction-refinement scheme, described in Chapter 4, develops a direct algorithm for solving the 3-valued model checking game. In case of an indefinite result, a cause for the indefinite result is identified by following the run of the algorithm that solves the game. This result also appears in [40].

In the second scheme, described in Chapter 5, a novel notion of a *non-losing* strategy is introduced and exploited for refinement. Here, the problem of solving the game is reduced to solving two 2-valued model checking (parity) games. In case the result is indefinite, the information needed for refinement is derived from the corresponding non-losing strategies. This approach is beneficial since it can use any solver for 2-valued parity games. Thus, it can take advantage of newly developed such algorithms with improved complexity. This result also appears in [41].

Our abstraction-refinement schemes are *incremental* in the sense that refinement is applied only where indefinite results exist and definite results from prior iterations are used within the model checking algorithm. For finite concrete models our abstraction-refinement schemes are fully automatic and guaranteed to terminate with a definite result true or false.

Monotonic Abstraction-Refinement In this work we develop a *monotonic* 3-valued abstraction-refinement framework for the μ-calculus, thus improving the effectiveness of the abstraction-refinement framework.

Monotonicity is an important aspect of abstraction-refinement. A refinement is called *monotonic* if the refined model is *more precise* in the sense that it satisfies more properties of the concrete model. In most 2-valued frameworks designed for universal logics such as ACTL and LTL, the refinement is monotonic. However, it turns out that when abstract models that combine both must and may transitions, such as KMTSs, are refined by splitting their states the refinement is not monotonic since formulas that had a definite value in the unrefined model may become indefinite. The problem lies in the must transitions which under-approximate the concrete transitions. We suggest to overcome the non-monotonicity problem by using must *hyper-transitions* to under-approximate the transitions of the system. A hyper-transition points to a *set* of states rather than a single state. Using such an abstract model, which we call *Generalized Kripke Modal Transition System* (GTS), ensures that the refinement is monotonic. Namely, it results in a more precise abstract model in which more formulas have a definite value. Yet, the number of hyper-transitions might be exponential in the number of states. To overcome this, we suggest an abstraction-refinement scheme where hyper-transitions are added gradually in each iteration. Model checking and refinement for GTSs are performed by a generalization of the game-based algorithms suggested for KMTSs. Thus, we obtain a monotonic game-based abstraction-refinement framework that is suitable for both verification and falsification of full μ-calculus. More details appear in Chapter 6.

These results also appear in [70], except that there the results are formulated for the logic CTL and the abstraction-refinement is based on a *symbolic* model checking algorithm instead of a game-based algorithm.

More Precision at Less Cost This work investigates both the precision and the model checking efficiency of abstract models designed to preserve the μ-calculus w.r.t. a 3-valued semantics.

We refer to precision measured w.r.t. the choice of abstract states, independently of the formalism used to describe abstract models. We show that the previous approach that uses GTSs as abstract models does not ensure *maximal precision.* We suggest a new class of 3-valued models for the μ-calculus, called *Hyper Kripke Modal Transition Systems* (HTSs), and a construction of an abstract HTS which is *most precise* w.r.t. *any* choice of abstract states. That is, given a set of abstract states and an abstraction mapping that relates each abstract state to the set of concrete states it represents, the constructed model is at least as precise as any other 3-valued model. HTSs use both must and may hyper-transitions. As in the case of GTSs, the construction of such models might involve an exponential blowup, which is inherent by the use of hyper-transitions. We therefore suggest an efficient algorithm for the alternation-free fragment of the μ-calculus in which the abstract HTS is constructed *during* model checking, by need. Our algorithm achieves maximal precision w.r.t. the given property while remaining quadratic in the number of abstract states. To complete the picture, we incorporate it into an abstraction-refinement framework. More details appear in Chapter 7. These results also appear in [72].

Compositional Verification and 3-Valued Abstractions Join Forces In this work, we join the forces of abstraction and compositional verification to obtain a novel fully automatic compositional technique that can determine the truth value of the full μ-calculus w.r.t. a given system.

Our approach is based on techniques taken from the game-based 3-valued model checking for abstract models. Namely, given a system $M = M_1 \| M_2$, we view each component M_i as an abstraction $M_i\!\uparrow$ of the global system. The abstract component $M_i\!\uparrow$ is defined using a 3-valued semantics so that whenever a μ-calculus formula φ has a definite value (true or false) on $M_i\!\uparrow$, the same value holds also for M. Thus, φ can be checked on either $M_1\!\uparrow$ or $M_2\!\uparrow$ (or both), and if any of them returns a definite result, then this result holds also for M. If both checks result in an indefinite value, the composition of the components needs to be considered. However, instead of constructing the composition of $M_1\!\uparrow$ and $M_2\!\uparrow$, we suggest how to automatically identify and compose only the parts of the components in which their composition is necessary in order to conclude the truth value of the formula at hand. The parts which can be handled separately are ignored. The resulting model is often significantly smaller than the full system. Furthermore, we explain how our compositional approach can be combined with abstraction, in order to further reduce the size of the checked components. The result is an incremental compositional abstraction-refinement framework, which resembles automatic Assume-Guarantee reasoning. More details appear in Chapter 8. These results also appear in [73].

Chapter 2

Preliminaries

2.1 The μ-calculus

Syntax We present our logic in negation normal form. Let AP be a finite set of atomic propositions and $\mathcal{V}$ a set of propositional variables. The set of literals over AP is defined to be $Lit = AP \cup \{\neg p : p \in AP\}$. We identify $\neg\neg p$ with p. The logic *μ-calculus* [50] in *negation normal form*[1] over AP is defined by the following grammar:

$$\varphi \ ::= \ l \ \mid \ \Box\varphi \ \mid \ \Diamond\varphi \ \mid \ \varphi \wedge \varphi \ \mid \ \varphi \vee \varphi \ \mid \ Z \ \mid \ \mu Z.\varphi \ \mid \ \nu Z.\varphi$$

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

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

We also consider the *alternation-free* fragment of the μ-calculus, denoted $\mathcal{L}^0_\mu$, where no nesting of fixpoints is allowed. Namely, $\varphi \in \mathcal{L}^0_\mu$ if for every subformula $\eta Z.\psi \in Sub(\varphi)$, no variable other than Z occurs freely in ψ.

Intuitively, $\Box$ stands for "all successors", whereas $\Diamond$ stands for "exists a successor". μ denotes a least fixpoint, whereas ν denotes greatest fixpoint. Least fixpoints are used to express liveness, while greatest fixpoints express safety properties. For example, the $\mathcal{L}_\mu$ fromula $\mu Z.(p \vee \Diamond Z)$ expresses the (existential) liveness property "there exists a path along which p eventually holds". Dually, $\nu Z.(p \wedge \Box Z)$ expresses the (universal) safety property "p is true in all the reachable states". These formulas are both alternation-free formulas. The semantics is formally defined in the following sections.

[1]This is without loss of generality, as every formula can be translated to an equivalent formula in negation normal form by pushing negations to the literals, while exchanging $\wedge$ with $\vee$, $\Box$ with $\Diamond$, and μ with ν.

2.2 Concrete Models and Concrete Semantics

Concrete systems are typically modelled as *Kripke structures.*

Definition 2.1. *[19] A* Kripke structure *is a tuple $M = (AP, S, S^0, R, L)$, where AP is a finite set of atomic propositions, S is a (finite) set of states, $S^0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \to 2^{Lit}$ is a labeling function, such that for every state s and every $p \in AP$, exactly one of p and $\neg p$ is in $L(s)$.*

AP is omitted whenever it is clear from the context. In some cases, when we are interested in a particular state, the set of initial states S^0 is omitted as well. In other cases, a single initial state, denoted $s^0 \in S$, is given instead of S^0. Unless explicitly stated otherwise, we consider finite-state models, in which S is a finite set. Note, that we do not require R to be *total*, i.e. we do not require that for every $s \in S$ there exists $t \in S$ such that sRt. Note further that we define the labeling function L as a mapping from states to sets of *literals* rather than atomic propositions. Typically, L maps each state to the set of atomic propositions that holds in it, with the meaning that if $p \notin L(s)$, then $\neg p$ holds in s. We explicitly add $\neg p$ to $L(s)$. This allows us a more unified presentation later when abstract models are introduced.

An example of a Kripke structure is depicted in Figure 2.1(a).

Concrete Semantics Let $\mathbb{B}$ be the complete boolean lattice consisting of elements $\{\text{tt}, \text{ff}\}$ denoting truth and falsity respectively, ordered by $\text{ff} \leq \text{tt}$. We use $\mathbb{B}$ to interpret the meaning of formulas of the μ-calculus over concrete Kripke structures.

The *concrete semantics* $[\![\varphi]\!]^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ over AP w.r.t. a Kripke structure $M = (AP, S, S^0, R, L)$ is an element of $S \to \{\text{tt}, \text{ff}\}$ – the functions from S to $\{\text{tt}, \text{ff}\}$ – which form a complete lattice under pointwise ordering: for $g, h : S \to \{\text{tt}, \text{ff}\}$, $g \sqsubseteq h$ iff $\forall s \in S : g(s) \leq h(s)$. Joins and meets in this lattice are denoted $g \sqcup h$ and $g \sqcap h$ resp.

To handle subformulas which are not closed, an *environment* $\rho : \mathcal{V} \to (S \to \{\text{tt}, \text{ff}\})$, which explains the meaning of free variables, is introduced. $[\![\varphi]\!]^{M,\rho}$ is defined inductively as follows. We denote with $\rho[Z \mapsto g]$ the environment that maps Z to g and agrees with ρ on all other arguments. In the following definition $f = \lambda g.[\![\varphi]\!]^{M,\rho[Z\mapsto g]}$ is an element of $(S \to \{\text{tt}, \text{ff}\}) \to (S \to \{\text{tt}, \text{ff}\})$ and *lfp*(f), *gfp*(f) stand for the least and greatest fixpoints of f. *lfp*$(f) = \bigsqcap \{g \mid [\![\varphi]\!]^{M,\rho[Z\mapsto g]} \sqsubseteq g\}$, and *gfp*$(f) = \bigsqcup \{g \mid g \sqsubseteq [\![\varphi]\!]^{M,\rho[Z\mapsto g]}\}$. These fixpoints exist according to [79], since the functions in $S \to \{\text{tt}, \text{ff}\}$ form a complete lattice when equipped with the partial order $\sqsubseteq$ defined above and the functional f is monotone w.r.t. the order $\sqsubseteq$ for any Z, φ and S.

$$
\begin{array}{lcl}
[\![l]\!]^{M,\rho} & := & \lambda s.\begin{cases} \mathrm{tt}, & \text{if } l \in L(s) \\ \mathrm{ff}, & \text{if } \neg l \in L(s) \end{cases} \\
[\![\Box\varphi]\!]^{M,\rho} & := & \lambda s.\begin{cases} \mathrm{tt}, & \text{if } \forall t \in S, \text{ if } sRt \text{ then } [\![\varphi]\!]^{M,\rho}(t) = \mathrm{tt} \\ \mathrm{ff}, & \text{if } \exists t \in S \text{ s.t. } sRt \text{ and } [\![\varphi]\!]^{M,\rho}(t) = \mathrm{ff} \end{cases} \\
[\![\Diamond\varphi]\!]^{M,\rho} & := & \lambda s.\begin{cases} \mathrm{tt}, & \text{if } \exists t \in S, \text{ s.t. } sRt \text{ and } [\![\varphi]\!]^{M,\rho}(t) = \mathrm{tt} \\ \mathrm{ff}, & \text{if } \forall t \in S \text{ if } sRt \text{ then } [\![\varphi]\!]^{M,\rho}(t) = \mathrm{ff} \end{cases} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{M,\rho} & := & [\![\varphi_1]\!]^{M,\rho} \sqcap [\![\varphi_2]\!]^{M,\rho} \\
[\![\varphi_1 \vee \varphi_2]\!]^{M,\rho} & := & [\![\varphi_1]\!]^{M,\rho} \sqcup [\![\varphi_2]\!]^{M,\rho} \\
[\![Z]\!]^{M,\rho} & := & \rho(Z) \\
[\![\mu Z.\varphi]\!]^{M,\rho} & := & \mathit{lfp}(\lambda g.[\![\varphi]\!]^{M,\rho[Z\mapsto g]}) \\
[\![\nu Z.\varphi]\!]^{M,\rho} & := & \mathit{gfp}(\lambda g.[\![\varphi]\!]^{M,\rho[Z\mapsto g]})
\end{array}
$$

For a closed formula φ, $[\![\varphi]\!]^{M,\rho} = [\![\varphi]\!]^{M,\rho'}$, for any environments ρ, ρ'. Thus, when closed formulas are considered, we drop the environment from the semantic brackets, and simply refer to $[\![\varphi]\!]^{M}$.

$[\![\varphi]\!]^{M}(s) = \mathrm{tt}$ $(= \mathrm{ff})$ means that the formula φ is true (false) in the state s of the Kripke structure M.

Similarly, M satisfies φ, denoted $M \models \varphi$, if for every $s^0 \in S^0$: $[\![\varphi]\!]^{M}(s^0) = \mathrm{tt}$. On the other hand, if there exists $s^0 \in S^0$ such that $[\![\varphi]\!]^{M}(s^0) = \mathrm{ff}$, then M falsifies φ, denoted $M \not\models \varphi$. In particular, if M has a single initial state s^0, then M satisfies (falsifies) φ if $[\![\varphi]\!]^{M}(s^0) = \mathrm{tt}$ $(= \mathrm{ff})$.

Approximants of $\mathcal{L}_\mu$ formulas are defined w.r.t. an environment ρ in the usual way: if $\mathit{fp}(Z) = \mu Z.\varphi$ then $Z^0_\rho := \lambda s.\mathrm{ff}$, $Z^{\alpha+1}_\rho := [\![\varphi]\!]^{M,\rho[Z\mapsto Z^\alpha_\rho]}$ for any ordinal α, and $Z^\lambda_\rho := \bigsqcup_{\alpha<\lambda} Z^\alpha_\rho$ for any limit ordinal λ.[2] Dually, if $\mathit{fp}(Z) = \nu Z.\varphi$ then $Z^0_\rho := \lambda s.\mathrm{tt}$, $Z^{\alpha+1}_\rho := [\![\varphi]\!]^{M,\rho[Z\mapsto Z^\alpha_\rho]}$, and $Z^\lambda_\rho := \bigsqcap_{\alpha<\lambda} Z^\alpha_\rho$.

The next theorem is a standard consequence of the Knaster-Tarski theorem [79].

Theorem 2.2. *For all Kripke structures M with state set S, and all environments ρ there is an ordinal α such that for all $s \in S$ we have:*

$$\text{if } [\![\eta Z.\varphi]\!]^{M,\rho}(s) = x \text{ then } Z^\alpha_\rho(s) = x.$$

2.3 Abstraction and 3-Valued Semantics

In this section we present abstract models for the μ-calculus and their relation to concrete models. It turns out that in order to guarantee preservation of μ-calculus formulas, which combine both universal and existential quantifiers, from abstract models to concrete models, we need to introduce *two* transition relations [54, 26]: preservation of truth of *universal* properties requires an over-approximation, whereas preservation of truth of *existential* properties requires an under-approximation. As such, *Kripke Modal Transition Systems* [45, 38], which contain both must and may transitions, are often used as abstract models that preserve the μ-calculus.

[2]Limit ordinals are only needed if the state space S of M is infinite.

Definition 2.3. *A* Kripke Modal Transition System *(KMTS) is a tuple $M = (AP, S, S^0, R^+, R^-, L)$, where AP, S, and S^0 are defined as before, $R^+, R^- \subseteq S \times S$ are must and may transition relations (resp.) such that $R^+ \subseteq R^-$, and $L : S \to 2^{Lit}$ is a labeling function such that for every state s and $p \in AP$,* at most *one of p and $\neg p$ is in $L(s)$.*

KMTSs generalize the notion of a Kripke structure in two ways. First, the requirement that exactly one of p and $\neg p$ is in $L(s)$ is relaxed by allowing that none of them is in $L(s)$. Second, two transition relations are introduced instead of R.

3-Valued Semantics The *3-valued semantics* $[\![\varphi]\!]_3^M$ of a closed formula $\varphi \in \mathcal{L}_\mu$ w.r.t. a KMTS M is a mapping from S to $\{\text{tt}, \text{ff}, \bot\}$. The 3-valued semantics [11, 45, 39][3] preserves both satisfaction (tt) and falsification (ff) from the abstract KMTS to the concrete model it represents. $\bot$ is a new truth value whose meaning is that the truth value over the concrete model is unknown and can be either tt or ff. The truth values tt and ff are also called *definite*, whereas $\bot$ is *indefinite*.

Here too, to handle subformulas which are not closed, an environment $\rho : \mathcal{V} \to (S \to \{\text{tt}, \text{ff}, \bot\})$, which explains the meaning of free variables, is introduced.

$[\![\varphi]\!]_3^{M,\rho}$ is defined inductively, similarly to $[\![\varphi]\!]^{M,\rho}$. The semantics of the logical operators $\wedge, \vee$ and of the fixpoints extends to the 3-valued case in a straightforward way by extending the ordering of the truth values to $\text{ff} \leq \bot \leq \text{tt}$, resulting in the lattice $\mathbb{B}_3$. The order $\sqsubseteq$ of the functions in $S \to \{\text{tt}, \text{ff}, \bot\}$ is extended accordingly. The semantics of the literals and the modalities is extended to the 3-valued case as follows.

$$
\begin{array}{lcl}
[\![l]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \text{tt}, & \text{if } l \in L(s) \\ \text{ff}, & \text{if } \neg l \in L(s) \\ \bot, & \text{otherwise} \end{cases} \\
[\![\Box\psi]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \text{tt}, & \text{if } \forall t \in S, \text{ if } sR^-t \text{ then } [\![\psi]\!]_3^{M,\rho}(t) = \text{tt} \\ \text{ff}, & \text{if } \exists t \in S \text{ s.t. } sR^+t \text{ and } [\![\psi]\!]_3^{M,\rho}(t) = \text{ff} \\ \bot, & \text{otherwise} \end{cases} \\
[\![\Diamond\psi]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \text{tt}, & \text{if } \exists t \in S, \text{ s.t. } sR^+t \text{ and } [\![\psi]\!]_3^{M,\rho}(t) = \text{tt} \\ \text{ff}, & \text{if } \forall t \in S \text{ if } sR^-t \text{ then } [\![\psi]\!]_3^{M,\rho}(t) = \text{ff} \\ \bot, & \text{otherwise} \end{cases}
\end{array}
$$

As in the concrete case, when only closed formulas are considered, we omit the environment from the semantics brackets. The notion of approximants and Theorem 2.2 carry over to the 3-valued case as well.

The intuition behind the 3-valued semantics for KMTSs lies in their application as abstract models, where must transitions under-approximate the concrete transitions, and may transitions over-approximate the concrete transitions. Namely, the 3-valued semantics is defined such that a formula is evaluated to tt or ff only when the abstract information suffices to determine such a definite truth value that will hold in the represented concrete model. Therefore, truth of universal formulas (of the form $\Box\psi$) is examined

[3]The 3-valued semantics for μ-calculus defined in [45] is given by means of two semantics, "necessarily" and "possibly", where a third possibility "not possibly" is obtained as the complement of the other two. When viewing "necessarily" as tt, "not possibly" as ff and "possibly but not necessarily" as $\bot$, this semantics coincides with ours.

along all the may transitions (which overapproximate the concrete transitions), whereas falsity of such formulas is shown by a must transition (which underapproximates the concrete transitions). Dually for existential formulas (of the form $\Diamond\psi$).

The notations $M \models \varphi$ and $M \not\models \varphi$ are used for a KMTS as well. In addition, if neither $M \models \varphi$ nor $M \not\models \varphi$ holds, then the value of φ in M is indefinite, denoted $M \overset{?}{\models} \varphi$.

Preservation of the μ-calculus The following definition formalizes the relation between two KMTSs that guarantees preservation of μ-calculus formulas w.r.t. the 3-valued semantics.

Definition 2.4 (Mixed Simulation)**.** *[26, 38] Let $M_1 = (AP, S_1, S_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP, S_2, S_2^0, R_2^+, R_2^-, L_2)$ be two KMTSs, both defined over AP. We say that $H \subseteq S_1 \times S_2$ is a* mixed simulation *from M_1 to M_2 if $(s_1, s_2) \in H$ implies the following:*

1. *$L_2(s_2) \subseteq L_1(s_1)$.*
2. *if $s_1 R_1^- s_1'$, then there is some $s_2' \in S_2$ s.t. $s_2 R_2^- s_2'$ and $(s_1', s_2') \in H$.*
3. *if $s_2 R_2^+ s_2'$, then there is some $s_1' \in S_1$ s.t. $s_1 R_1^+ s_1'$ and $(s_1', s_2') \in H$.*

If there is a mixed simulation H s.t. $(s_1, s_2) \in H$, then $(M_1, s_1) \preceq (M_2, s_2)$.

If there is a mixed simulation H s.t. $\forall s_1^0 \in S_1^0\ \exists s_2^0 \in S_2^0$ s.t. $(s_1^0, s_2^0) \in H$, and $\forall s_2^0 \in S_2^0\ \exists s_1^0 \in S_1^0$ s.t. $(s_1^0, s_2^0) \in H$, then M_2 is greater by the mixed simulation relation *than M_1, denoted $M_1 \preceq M_2$.*

Definition 2.4 can be applied to a (concrete) Kripke structure M_C and an (abstract) KMTS M_A, by viewing the Kripke structure as a KMTS where $R^+ = R^- = R$. For a Kripke structure, the 3-valued semantics agrees with the concrete semantics. Thus, preservation of $\mathcal{L}_\mu$ formulas from an abstract model to the concrete model is guaranteed by the following theorem.

Theorem 2.5. *[38] Let $H \subseteq S_1 \times S_2$ be a mixed simulation relation from a KMTS M_1 to a KMTS M_2. Then for every $(s_1, s_2) \in H$ and every $\varphi \in \mathcal{L}_\mu$ we have that $[\![\varphi]\!]_3^{M_2}(s_2) \neq \perp \Rightarrow [\![\varphi]\!]_3^{M_1}(s_1) = [\![\varphi]\!]_3^{M_2}(s_2)$.*

We conclude that if $M_1 \preceq M_2$, then for every $\varphi \in \mathcal{L}_\mu$:

- *$M_2 \models \varphi \Rightarrow M_1 \models \varphi$, and*
- *$M_2 \not\models \varphi \Rightarrow M_1 \not\models \varphi$.*

Note that the requirement of Definition 2.4 regarding the bi-directional correspondence between the initial states of the KMTSs M_1 and M_2 is needed in order to preserve both truth and falsity from M_2 to M_1 in the model-level whenever $M_1 \preceq M_2$. Namely, suppose $M_2 \models \varphi$. Then for every $s_2^0 \in S_2^0$: $[\![\varphi]\!]_3^{M_2}(s_2^0) = \mathrm{tt}$. In order to conclude that $[\![\varphi]\!]_3^{M_1}(s_1^0) = \mathrm{tt}$ holds for every $s_1^0 \in S_1^0$ as well, which ensures that $M_1 \models \varphi$, we require that every $s_1^0 \in S_1^0$ has a corresponding $s_2^0 \in S_2^0$ s.t. $(s_1^0, s_2^0) \in H$. On the other hand, suppose $M_2 \not\models \varphi$, which means that there exists $s_2^0 \in S_2^0$ s.t. $[\![\varphi]\!]_3^{M_2}(s_2^0) = \mathrm{ff}$. In order to conclude that there exists $s_1^0 \in S_1^0$ s.t. $[\![\varphi]\!]_3^{M_1}(s_1^0) = \mathrm{ff}$, which ensures that $M_1 \not\models \varphi$, we require that $s_1^0 \in S_1^0$ has a corresponding $s_2^0 \in S_2^0$ in H.

Abstraction Let M_C be a concrete Kripke structure with a set of concrete states S_C. We consider *state abstractions* that are performed by collapsing sets of concrete states (from S_C) into single abstract states (in S_A) via a concretization function γ. As such, an *abstraction* (S_A, γ) for S_C consists of a finite set of *abstract states* S_A and a total *concretization function* $\gamma : S_A \to 2^{S_C}$ that maps each abstract state to the (nonempty) set of concrete states it represents. Every $s_c \in S_C$ is represented by some $s_a \in S_A$.

Construction of an Abstract KMTS Given an abstraction (S_A, γ) for the set of states S_C of a (concrete) Kripke structure $M_C = (S_C, S_C^0, R, L_C)$, an abstract model, in the form of a KMTS $M_A = (S_A, S_A^0, R^+, R^-, L_A)$ which is greater by the mixed simulation relation than M_C, can be defined as follows.

The set of initial abstract states S_A^0 is built s.t.

$$s_a^0 \in S_A^0 \iff \exists s_c^0 \in S_C^0 \text{ s.t. } s_c^0 \in \gamma(s_a^0).$$

The "iff" is needed in order to ensure the two requirements of Definition 2.4 regarding the initial states. More specifically, "$\Longleftarrow$" is needed in order to preserve truth from M_A to M_C, while "$\Longrightarrow$" is needed to preserve falsity. This requirement is not needed for state-wise preservation, i.e., for preservation of properties from abstract states to the concrete states represented by them.

The labeling of an abstract state is defined in accordance with the labeling of all the concrete states it represents. Namely, for $l \in Lit$,

$$l \in L_A(s_a) \implies \forall s_c \in \gamma(s_a) : l \in L_C(s_c).$$

It is thus possible that neither p nor $\neg p$ are in $L_A(s_a)$.

The *may* transitions in an abstract model provide an over-approximation of the concrete transitions. They are computed by an $[\exists\exists]$ rule s.t. every concrete transition is represented by them:

$$\exists s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(s'_a) \text{ s.t. } s_c R s'_c \implies s_a R^- s'_a$$

Note that the implication allows that there will be additional may transitions as well. The *must* transitions, on the other hand, provide an under-approximation of the concrete transitions. They represent concrete transitions that are common to all the concrete states represented by the source abstract state. They are computed by a $[\forall\exists]$ rule:

$$\forall s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(s'_a) \text{ s.t. } s_c R s'_c \Longleftarrow s_a R^+ s'_a$$

Note that it is possible that there are less must transitions than allowed by this rule.

That is, unlike the initial states, the labeling, may and must transitions do not have to be exact, as long as they maintain these conditions.

Exact KMTS If the three implications used in the definition of the labeling and the transitions are replaced by "iff", then the labeling, may and must transitions are *exact*, resulting in the *exact KMTS*. This model is *most precise* compared to all the KMTSs that are constructed as described above w.r.t. the given abstraction (S_A, γ).

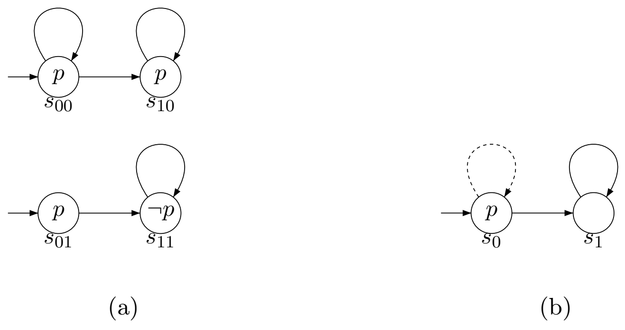


Figure 2.1: (a) A concrete Kripke structure, and (b) an abstract KMTS for it.

Example 2.6. Consider the concrete system shown in Figure 2.1(a), employing a single atomic proposition p. Joining s_{00} and s_{01} and respectively s_{10} and s_{11} yields the (exact) KMTS shown in Figure 2.1(b), where *may* transitions are shown as dotted arrows only when no *must* transitions are present. Note that the state s_1 is not labeled by p, nor by $\neg p$, which means that the value of p in it is indefinite.

Other constructions of abstract models can be used as well. For example, if γ is a part of a *Galois Connection* [24] ($\gamma : \mathcal{S}_A \to 2^{\mathcal{S}_C}, \alpha : 2^{\mathcal{S}_C} \to \mathcal{S}_A$) from $(2^{\mathcal{S}_C}, \subseteq)$ to $(\mathcal{S}_A, \sqsubseteq)$, then an abstract model can be constructed as described in [26] within the framework of *Abstract Interpretation* [24, 61, 26]. It is then not guaranteed that the must transitions are a subset of the may transitions, which complicates our further developments. In this work we always assume that the must transitions are a subset of the may transitions.

All the above constructions assure us that whenever $s_c \in \gamma(s_a)$, then $(M_C, s_c) \preceq (M_A, s_a)$. The mixed simulation $H \subseteq S_C \times S_A$ is induced by γ as follows: $(s_c, s_a) \in H$ iff $s_c \in \gamma(s_a)$. Moreover, the definition of the initial abstract states ensures that $M_C \preceq M_A$. Therefore, Theorem 6.7 guarantees preservation of $\mathcal{L}_\mu$ from M_A to M_C.

Chapter 3

3-Valued Model Checking Games

3.1 Introduction

In this chapter, we present a new model checking game for abstract models (KMTSs) w.r.t. specifications in the μ-calculus, interpreted over a 3-valued semantics. This sets the basis for the game-based 3-valued model checking and abstraction-refinement presented in the following chapters.

Many algorithms for μ-calculus model checking w.r.t. the concrete (2-valued) semantics have been suggested in the literature [33, 78, 80, 21, 62]. An elegant solution to this problem is the game-based approach [76], in which two players, the verifier (denoted $\exists$) and the falsifier (denoted $\forall$), try to win a game. A formula φ is true in a model M iff the verifier has a winning strategy, meaning that she can win any play, no matter what the falsifier does. Otherwise, the falsifier has a winning startegy and the formula is false in the model. The game is played on a *game board*, consisting of configurations $s \vdash \psi$, where s is a state of the model M and ψ is a subformula of φ. The players make moves between configurations in which they try to verify or falsify ψ in s. These games can also be seen and studied as *parity games* [32, 47] and we follow this approach as well.

In model checking games for the 2-valued semantics, exactly one of the players has a winning strategy, thus the model checking result is either true or false. For the 3-valued semantics, a third value should also be possible. Following our previous work [74] for the logic CTL, we change the definition of a game for the μ-calculus so that a *tie* is also possible. As before, Player $\exists$ has a winning strategy iff $M \models \varphi$, and Player $\forall$ has a winning strategy iff $M \not\models \varphi$. However, it is also possible that neither of them has a winning strategy, in which case the value of φ in M is *indefinite*. To simplify the presentation, we introduce 3-valued parity games as a generalization of ordinary (2-valued) parity games, and transform the 3-valued model checking game into an equivalent 3-valued parity game with players 0 and 1.

In the following chapters we use the 3-valued model checking game, in its formulation as a 3-valued parity game, as the basis for a model checking algorithm, as well as a refinement mechanism.

3.1.1 Related work

Our work characterizes the 3-valued model checking problem of the μ-calculus in terms of two-players games. This generalizes the characterization of the model checking problem via the game-theoretic approach in the concrete case [75].

A different characterization of the (concrete) model checking problem can be given in terms of *alternating automata* as part of the automata-theoretic approach to model checking [10, 51]. Alternating automata generalize the standard notion of nondeterministic automata by allowing several successor states in the automaton to simultaneously proceed along the input. In the automata-theoretic approach of [10, 51], an alternating (tree) automaton whose language is the set of all trees that satisfy the formula is constructed. It "describes" all the models that satisfy the given formula. The alternating automaton is assembled with the given system model, resulting in the product automaton, which is also an alternating (word) automaton. Model checking is then performed by checking nonemptiness of the product automaton, which represents the product of the model and the checked formula.

The game-based and the automata-based approaches to model checking have a strong resemblance in the concrete setting (see e.g. [58]): Similarly to the configurations of the model checking game, each state of the automaton represents a state of the model and a subformula of the checked formula. Furthermore, a winning strategy of the verifier in the model checking game corresponds to an accepting run of the product automaton and vice versa. Our work can also be developed in the automata-theoretic framework, yet this would require some generalization of alternating automata.

The 3-valued model checking game developed in this chapter can be viewed as a special case of the multi-valued model checking game that we suggested in a more recent work [71]. In multi-valued model checking, the labeling of states and the transitions of the system, as well as the meaning of formulas, are interpreted as elements from a lattice. In the general case of multi-valued model checking games, we no longer talk about winning. Instead, we talk about the *value* of the game, which is an element from the lattice. This is unlike the 3-valued case, where the truth values can still be encoded as winning with the additional possibility of a tie, which corresponds to the truth value $\perp$. This makes the 3-valued case unique compared to the use of an arbitrary lattice.

Further related work regarding the resulting game-based 3-valued model checking algorithm, as well as the abstraction-refinement framework, appears in Chapter 4.

3.2 Model Checking Games for 3-Valued μ-Calculus

The *3-valued model checking game* $\Gamma_M(s_0, \varphi_0)$ on a KMTS[1] $M = (S, R^+, R^-, L)$ with state $s_0 \in S$ and a $\mathcal{L}_\mu$ formula φ_0 is played by Players $\exists$ and $\forall$ in order to determine the truth value of φ_0 in s_0, cf. [75]. Player $\exists$ has the role of the *verifier*, while Player $\forall$ takes the role of the *falsifier*.

[1]We do not consider a set of initial states in the KMTS since the game is defined for a particular state. This state can be viewed as an initial state.

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

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

$$\frac{s \vdash \Diamond\varphi}{t \vdash \varphi}\ \exists:\ sR^{+}t \text{ or } sR^{-}t \qquad \frac{s \vdash \Box\varphi}{t \vdash \varphi}\ \forall:\ sR^{+}t \text{ or } sR^{-}t$$

Figure 3.1: The model checking game rules for 3-valued μ-calculus.

Configurations, Moves and Plays Configurations are elements of $\mathcal{C} \subseteq S \times \mathit{Sub}(\varphi_0)$, and written $t \vdash \psi$. Each play of $\Gamma_M(s_0, \varphi_0)$ is a maximal sequence of configurations that starts with $s_0 \vdash \varphi_0$. The game rules are presented in Figure 3.1. Each rule is marked by $\exists$ / $\forall$ to indicate which player makes the move. A rule is applied when the player is in configuration C_i, which is of the form of the upper part of the rule. C_{i+1} is then the configuration in the lower part of the rule. The rules shown in the first and third lines present a choice which the player can make. Since no choice is possible when applying the rules shown in the second line, we arbitrarily assign one player, let us say $\exists$, and call the rules *deterministic*. If no rule can be applied, the play terminates.

The configurations of $\Gamma_M(s_0, \varphi_0)$ are classified as $\wedge$, $\vee$, $\Box$, $\Diamond$, or literal configurations, based on their subformuals. Configurations whose subformulas are of the form Z or $\eta Z.\psi$ are called *deterministic*. Configurations where the play terminates are also called *terminal configurations*. These are either literal configurations or $\Box$ and $\Diamond$ configurations in which the corresponding state of the model has no outgoing transitions.

The reachable configurations in the game present all the information "relevant" for the model checking: Intuitively, a reachable configuration of the form $t \vdash \psi$ indicates that the value of the subformula ψ in the state t is relevant for determining the value of φ_0 in s_0. The moves that the players choose from a configuration $t \vdash \psi$ present "subgoals" that the players define for verifying or falsifying ψ in t. For example, in a configuration of the form $t \vdash \psi_0 \vee \psi_1$, Player $\exists$, the verifier, chooses the subformula that she intends to verify in t. In a configuration of the form $t \vdash \Diamond\psi$, she chooses a successor of t in which she intends to verify ψ. In configurations of the form $t \vdash \psi_0 \wedge \psi_1$ and $t \vdash \Box\psi$, Player $\forall$, the falsifier, makes similar choices for falsification. In order to be able to both verify and falsify each subformula, the game allows the players to use both may and must transitions in $\Diamond$ and $\Box$ configurations. The reason is that, for example, truth of a formula $\Box\psi$ in a state t should be checked upon outgoing may transitions of t, but its falsity should be checked upon outgoing must transitions.

Winning Conditions The players use must transitions in order to win, while they use may transitions in order to prevent the other player from winning. To demonstrate this, consider again the $\Box\psi$ example: in a configuration of the form $t \vdash \Box\psi$, Player $\forall$, the falsifier, makes a move. If he wants to falsify $\Box\psi$, he needs to show a must transition from t to a state that falsifies ψ. Yet, if he only wishes to prevent Player $\exists$ from verifying ψ, then it suffices to show a may transition to a state that does not satisfy ψ.

Thus, a player can only win the play if he or she is eager in their moves, meaning that the player always makes moves that are designed for verification (if the player is Player $\exists$), or always makes moves that are designed for falsification (if it is Player $\forall$). These moves are all based on must transitions. The other player, on the other hand, possibly uses both types of transitions:

Definition 3.1. *A player is said to play* eagerly[2] *if she or he never chooses a transition of type $R^- \setminus R^+$. A play is called* $\exists$-eager*, resp.* $\forall$-eager*, if Player $\exists$, resp. Player $\forall$, plays eagerly.*

As a result of the eagerness requirements, it is possible that none of the players wins a play, i.e. the play ends with a tie. A tie can also occur due to the 3-valued nature of the labeling function of the KMTS, by reaching a literal configuration where the value of the literal is indefinite. More precisely:

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

1. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash l$ with $l \in L(s)$, or
2. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash \Box\psi$ and there is no $t' \in S$ s.t. tR^-t', or
3. the outermost variable that occurs infinitely often is of type ν.

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

4. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash l$ with $\neg l \in L(s)$, or
5. there is an $n \in \mathbb{N}$, s.t. $C_n = t \vdash \Diamond\psi$ and there is no $t' \in S$ s.t. tR^-t', or
6. the outermost variable that occurs infinitely often is of type μ.

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

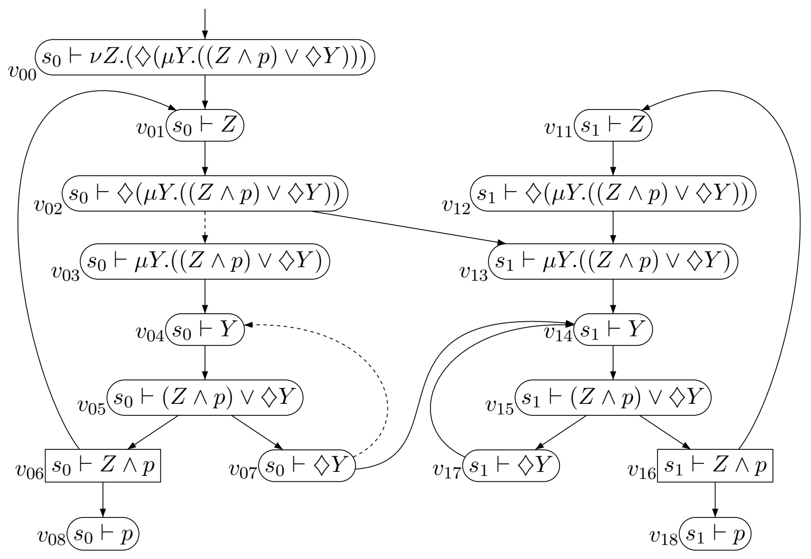
Example 3.2. For the model checking problem of the formula

$$\nu Z.(\Diamond(\mu Y.((Z \wedge p) \vee \Diamond Y)))$$

in the state s_0 of the abstract KMTS from Figure 2.1(b) all possible moves are shown in Figure 3.2. Configurations in which $\exists$ is to choose are drawn as circles while $\forall$-configurations are shown as squares. Moves based on *may* transitions are not shown when the same move is possible using a *must* transition.

The result of the play $v_{00}v_{01}v_{02}v_{03}v_{04}v_{05}v_{06}v_{08}$ is a tie: while the play ends in a configuration in which p evaluates to tt, $\exists$ does not win since choosing the edge from v_{02} to v_{03} violates $\exists$-eagerness. $v_{00}v_{01}v_{02}v_{13}v_{14}v_{15}v_{16}v_{11}v_{12}v_{13}\ldots$, on the other hand, is an $\exists$-eager play in which the outermost variable occurring infinitely often is of type ν. Thus, it is won by $\exists$.

[2]The notion of 'eagerness' replaces the notion of 'consistency' from [74, 40].



v_{00} $s_0 \vdash \nu Z.(\Diamond(\mu Y.((Z \wedge p) \vee \Diamond Y)))$
v_{01} $s_0 \vdash Z$
v_{02} $s_0 \vdash \Diamond(\mu Y.((Z \wedge p) \vee \Diamond Y))$
v_{03} $s_0 \vdash \mu Y.((Z \wedge p) \vee \Diamond Y)$
v_{04} $s_0 \vdash Y$
v_{05} $s_0 \vdash (Z \wedge p) \vee \Diamond Y$
v_{06} $s_0 \vdash Z \wedge p$
v_{07} $s_0 \vdash \Diamond Y$
v_{08} $s_0 \vdash p$
v_{11} $s_1 \vdash Z$
v_{12} $s_1 \vdash \Diamond(\mu Y.((Z \wedge p) \vee \Diamond Y))$
v_{13} $s_1 \vdash \mu Y.((Z \wedge p) \vee \Diamond Y)$
v_{14} $s_1 \vdash Y$
v_{15} $s_1 \vdash (Z \wedge p) \vee \Diamond Y$
v_{17} $s_1 \vdash \Diamond Y$
v_{16} $s_1 \vdash Z \wedge p$
v_{18} $s_1 \vdash p$

Figure 3.2: All possible moves in the game over the KMTS from Figure 2.1(b).

Strategies In order to relate the model checking game to the model checking problem, we need the notion of a strategy:

A *strategy* for Player q is a partial function $\zeta : \mathcal{C} \to \mathcal{C}$, such that its domain is the set of configurations where Player q moves and for all configurations C and C': $\zeta(C) = C'$ implies that there is a move from C to C'. Player q plays a game according to a strategy ζ if all his choices agree with ζ. A strategy for Player q is called a *winning strategy* if Player q wins every play where he plays according to this strategy. Note that we restrict ourselves to so-called *memoryless* strategies. This will be justified in Section 4.2.

Correctness The following theorem, proven in the remainder of this section, formalizes the relation between the model checking game $\Gamma_M(s_0, \varphi_0)$ and the truth value of φ_0 in the state s_0 of M.

Theorem 3.3. *Given a KMTS $M = (S, R^+, R^-, L)$, an $s_0 \in S$, and a closed $\varphi_0 \in \mathcal{L}_\mu$, we have:*

(a) $[\![\varphi_0]\!]^M(s_0) = \text{tt}$ iff Player $\exists$ has a winning strategy for $\Gamma_M(s_0, \varphi_0)$,

(b) $[\![\varphi_0]\!]^M(s_0) = \text{ff}$ iff Player $\forall$ has a winning strategy for $\Gamma_M(s_0, \varphi_0)$,

(c) $[\![\varphi_0]\!]^M(s_0) = \perp$ iff neither Player $\exists$ nor Player $\forall$ has a winning strategy for $\Gamma_M(s_0, \varphi_0)$.

Corollary 3.4. *Let $M_C = (S_C, R, L_C)$ be a Kripke structure with $s_c \in S_C$ and $M_A = (S_A, R^+, R^-, L_A)$ be an abstract model of M_C w.r.t. an abstraction (S_A, γ). Let $s_a \in S_A$ with $s_c \in \gamma(s_a)$ and $\varphi \in \mathcal{L}_\mu$. Then,*

(a) If Player $\exists$ has a winning strategy for $\Gamma_{M_A}(s_a, \varphi)$ then $[\![\varphi]\!]^{M_C}(s_c) = \text{tt}$.

(b) If Player $\forall$ has a winning strategy for $\Gamma_{M_A}(s_a, \varphi)$ then $[\![\varphi]\!]^{M_C}(s_c) = \mathrm{ff}$.

Proof. Suppose Player $\exists$ has a winning strategy in $\Gamma_{M_A}(s_a, \varphi)$. According to Theorem 3.3, we have $[\![\varphi]\!]^{M_A}(s_a) = \mathrm{tt}$. Applying Theorem 2.5, we get $[\![\varphi]\!]^{M_C}(s_c) = \mathrm{tt}$. Part (b) is proved analogously. □

The remainder of this section is devoted to the proof of Theorem 3.3. Intuitively, the theorem holds since the conditions for Player $\exists$, the verifier, having a winning strategy resemble the 3-valued semantics for truth. Dually, the conditions for Player $\forall$, the falsifier, having a winning strategy resemble the 3-valued semantics for falsity. Therefore, the player that has a winning strategy in the game corresponds to the truth value of the formula. For example, $\vee$ and $\Diamond$ configurations are controlled by Player $\exists$. Thus, Player $\exists$ has a winning strategy from such a configuration iff she has a winning strategy from at least *one* of the succeeding configurations along must transitions. However, in a $\wedge$ or a $\Box$ configuration, Player $\forall$ controls the moves and therefore Player $\exists$ has a winning strategy in such a configuration iff she has a winning strategy from *all* of the succeeding configurations along may transitions. This resembles the 3-valued semantics for truth of $\vee$, $\Diamond$, $\wedge$, and $\Box$ formulas respectively. We now turn to formally prove the theorem. In the following, we sometimes omit M from the semantics brackets.

Definition 3.5. *The* truth value *of a configuration $t \vdash \psi$ in the context of ρ is the value of $[\![\psi]\!]^{\rho}(t)$. The value* tt improves *both $\perp$ and* ff, *while $\perp$ only improves* ff. *On the other hand, x* worsens *y iff y improves x. A configuration with truth value x under the environment ρ will also be called an x_ρ-configuration. We say that a move, i.e. an application of a game rule between configurations C and C' is a x_ρ-$y_{\rho'}$-*improvement *if C is an x_ρ-configuration, C' is an $y_{\rho'}$-configuration and y improves x. Similarly we define a x_ρ-$y_{\rho'}$-*worsening *and a x_ρ-$y_{\rho'}$-preservation. In the latter case we obviously have $x = y$.*

An inspection of the game rules and the semantics together with Theorem 2.2, which holds in the case of KMTSs as well, proves the following.

Lemma 3.6. *For all environments ρ, and all truth values x, y we have:*

a) Player $\exists$ cannot eagerly make a move that is a x_ρ-y_ρ-improvement, but can always eagerly make a tt_ρ-tt_ρ-preservation.

b) Player $\forall$ cannot eagerly make a move that is a x_ρ-y_ρ-worsening, but can always eagerly make a ff_ρ-ff_ρ-preservation.

c) There is an ordinal α such that the deterministic rule for a fixpoint formula $\eta Z.\varphi$ is a x_ρ-$x_{\rho'}$-preservation when $\rho' := \rho[Z \mapsto Z^\alpha_\rho]$.

d) Let Z be an η-variable and $\rho(Z) = Z^\alpha_{\rho_i}$ for some environment ρ_i and some ordinal $\alpha > 0$. There is an ordinal $\beta < \alpha$ such that the deterministic rule for unfolding Z is an x_ρ-$x_{\rho'}$-preservation when $\rho' := \rho[Z \mapsto Z^\beta_{\rho_i}]$.

Proof. The proof is by a case analysis which considers for each configuration and possible truth value both the 3-valued semantics and the possible moves of the player. For example, consider a configuration of the form $t \vdash \psi_0 \vee \psi_1$, where Player $\exists$ moves. If its truth value in the context of ρ is tt, then by the definition of the semantics, the truth value of either $t \vdash \psi_0$ or $t \vdash \psi_1$ (or both) in the context of ρ is also tt. Player $\exists$ can therefore eagerly make a tt_ρ-tt_ρ-preservation by moving to the corresponding configuration whose truth value is tt. On the other hand, if the truth value of $t \vdash \psi_0 \vee \psi_1$ in the context of ρ is ff, then the truth value of both $t \vdash \psi_0$ and $t \vdash \psi_1$ in the context of ρ is also ff, which means that no matter which of them Player $\exists$ chooses, she will not be able to make an x_ρ-y_ρ-improvement. □

Lemma 3.7. *Let $\varphi \in \mathcal{L}_\mu$. Player $\exists$ does not have a winning strategy for the game $\Gamma_M(s, \varphi)$ if $[\![\varphi]\!]^M(s) \neq \text{tt}$.*

Proof. Suppose that on one hand $[\![\varphi]\!]^M(s) \neq \text{tt}$ but Player $\exists$ has a winning strategy ζ for $\Gamma_M(s, \varphi)$. Take the partial game tree induced by this strategy, i.e. the tree of all plays in which all of Player $\forall$'s choices are preserved but only those of Player $\exists$'s choices which agree with ζ.

First we show that this tree contains at least one play $C_0, C_1, \ldots$ for which there is a corresponding sequence of environments $\rho_0, \rho_1, \ldots$ such that for all $i \in \mathbb{N}$, C_i is not a tt_{ρ_i}-configuration. Let ρ_0 be the empty environment. Since φ is closed, the root of this tree is not a tt_{ρ_0}-configuration. According to Lemma 3.6, deterministic rules preserve the truth value of a configuration – possibly extending the environments – and Player $\exists$'s choices do not improve the truth value when considering the same environment, as she is playing eagerly (being that ζ is a winning strategy for her). This can only be done by Player $\forall$. However, suppose there is a configuration C_i in which Player $\forall$ makes a choice and which has a truth value other than tt under ρ_i. Then C_i is of the form $t \vdash \psi_0 \wedge \psi_1$ or $t \vdash \Box\psi$. For the former case note that the truth value of C_i under ρ_i is the infimum in $\mathbb{B}_3$ of its two successor's truth values under ρ_i. Thus, it is not tt only if there is a successor which has a truth value other than tt under the same environment ρ_i – which will also define ρ_{i+1} (that is, $\rho_{i+1} = \rho_i$). For the latter case consider $[\![\Box\psi]\!]^{\rho_i}(t)$. It can only differ from tt if there is a t' such that $tR^- t'$ and $[\![\psi]\!]^{\rho_i}(t') \neq \text{tt}$. But $t' \vdash \psi$ is a possible successor configuration of C_i. Thus, it is included in the tree and has a truth value which is not tt under ρ_i – which will again define ρ_{i+1} (i.e., $\rho_{i+1} = \rho_i$).

This argument can be iterated yielding a path $C_0, C_1, \ldots$ and a sequence $\rho_0, \rho_1, \ldots$ in which $\rho_i \neq \rho_{i+1}$ only if the move from C_i to C_{i+1} is deterministic. $C_0, C_1, \ldots$ is a path on which no tt-configuration under the according ρ_i occurs. Now, this path can either be finite or infinite. The first case immediately leads to a contradiction since finite paths won by Player $\exists$ necessarily end in a tt-configuration under any environment.

Suppose therefore that the path represents a play which is won by Player $\exists$'s winning condition 3. Then there is an outermost variable Z of fixpoint type ν which occurs infinitely often in this play. Take the last occurrence of a configuration $t \vdash \nu Z.\varphi$ and name it C_i. By assumption, $[\![\nu Z.\varphi]\!]^{\rho_i}(t) = x$ for some $x \neq \text{tt}$ and ρ_i as constructed above.

According to Lemma 3.6, ρ_{i+1} interprets Z as an approximant with some index $\alpha \neq 0$. That is, $\rho_{i+1}(Z) = Z^\alpha_{\rho_i}$. Lemma 3.6 also shows that subsequent environments ρ_j, $j > i$ interpret Z as approximants $Z^\beta_{\rho_i}$ with decreasing indices β. But the ordinals are well-founded. Hence, there is a j such that $C_j = t \vdash Z$ for some t and $\rho_j(Z) = Z^0_{\rho_i}$, meaning that $[\![Z]\!]^{\rho_j}(t) = \text{tt}$. But on the other hand, since C_j appears on the above path, we know that C_j is not a tt_{ρ_j}-configuration. This is a contradiction. We conclude that Player $\exists$ cannot have a winning strategy. $\square$

Lemma 3.8. *Let $\varphi \in \mathcal{L}_\mu$. Player $\exists$ has a winning strategy for the game $\Gamma_M(s,\varphi)$ if $[\![\varphi]\!]^M(s) = \mathrm{tt}$.*

Proof. Suppose $[\![\varphi]\!]^M(s) = \mathrm{tt}$. According to Lemma 3.6, Player $\exists$ can play eagerly in such a way that every reached configuration has truth value tt under some environment which is constructed successively using Lemma 3.6 and starting with the empty environment. Note that φ is assumed to be closed.

Player $\forall$ cannot help but to make moves that result in tt_ρ-configurations under the corresponding ρ as well. This defines a strategy for Player $\exists$. It remains to be seen that this strategy guarantees her to win every resulting play. First, by Lemma 3.6 again, every resulting play is $\exists$-eager. By preservation of the truth value, a finite play must end in a tt-configuration under an irrelevant environment. But then it is won by Player $\exists$ with winning condition 1 or 2.

Suppose therefore that the play $C_0, C_1, \ldots$ at hand is of infinite length. By Player $\exists$'s strategy that uses the construction of environments in Lemma 3.6, there are $\rho_0, \rho_1, \ldots$, such that C_i is a tt_{ρ_i}-configuration for all $i \in \mathbb{N}$.

Any infinite play has a unique outermost variable Z that occurs infinitely often, cf. [75]. This variable has a unique fixpoint type $\eta \in \{\mu, \nu\}$. Assume for the sake of contradiction that $fp(Z) = \mu Z.\psi$ for some ψ. Then take the last occurrence of a configuration $C_i = t \vdash \mu Z.\psi$. Since Z is outermost, it is guaranteed to exist, for otherwise there would be another fixpoint formula that generated $\mu Z.\psi$ infinitely often.

According to the construction of the strategy, there is an ordinal α such that ρ_{i+1} interprets Z in the following configuration $C_{i+1} = t \vdash Z$ by $Z^\alpha_{\rho_i}$. Again, by the construction of the strategy using Lemma 3.6, the next time Z occurs it is interpreted as $Z^\beta_{\rho_i}$ for some $\beta < \alpha$. By the well-foundedness of the ordinals, there will eventually be a tt_{ρ_k}-configuration $C_k = t' \vdash Z$ such that $\rho_k(Z) = Z^0_{\rho_i}$ which is impossible since $Z^0_{\rho_i} = \lambda s.\mathrm{ff}$, provided that the fixpoint type of Z is μ. Thus, the fixpoint type of Z must have been ν which makes Player $\exists$ the winner of the play at hand. $\square$

Lemma 3.9. *Let $\varphi \in \mathcal{L}_\mu$. Player $\forall$ has a winning strategy for the game $\Gamma_M(s,\varphi)$ iff $[\![\varphi]\!]^M(s) = \mathrm{ff}$.*

Proof. This is the dual to Lemmas 3.7 and 3.8. Hence, it is proved in the same way exchanging tt and ff, "improve" and "worsen", ν and μ, Player $\exists$ and $\forall$. $\square$

We can now return to the proof of Theorem 3.3:

Proof of Theorem 3.3. Parts (a) and (b) are proved in Lemmas 3.7, 3.8 and 3.9. For part (c) suppose that $[\![\varphi_0]\!]^M(s_0) = \perp$. Then none of the players can have a winning strategy because using parts (a) and (b) one would immediately contradict the assumption. Conversely, suppose that none of them has a winning strategy but $[\![\varphi_0]\!]^M(s_0) \neq \perp$. Again, using (a) or (b) one obtains an immediate contradiction. $\square$

3.3 3-Valued Parity Games

The previous section related the 3-valued model checking games with the 3-valued semantics of $\mathcal{L}_\mu$. For the sake of readability it is sometimes more convenient to deal with parity games. In the context of parity games, instead of Player $\exists$ and $\forall$, we talk of Player 0 and Player 1, resp., and use σ to denote Player 0 or 1 and $\overline{\sigma} = 1 - \sigma$ for the opponent[3].

Parity games are traditionally used to describe the model checking game for the μ-calculus [32]. For simplicity, we consider parity games with dead-end vertices (see Remark 3.14). In order to describe our 3-valued game for $\mathcal{L}_\mu$, we need to generalize parity games in the following ways: (1) we have two types of edges: must edges and may edges, where every must edge is also a may edge, (2) terminal configurations (dead-ends) are classified as either winning for one player, or as tie-configurations, and (3) an eagerness requirement is added to the winning conditions.

3.3.1 3-Valued Parity Games

A *3-valued parity game* $\mathcal{G} = (A, \Theta)$ has an *arena* $A = (V_0, V_1, V_{tie}, E^+, E^-)$ such that V_0, V_1 and V_{tie} are disjoint sets of vertices. Let $V := V_0 \cup V_1 \cup V_{tie}$. Then $E^+ \subseteq E^- \subseteq (V \setminus V_{tie}) \times V$ are sets of must and may edges, meaning that every $v \in V_{tie}$ is a dead-end. Edges in $E^- \setminus E^+$ are sometimes called genuine may edges. $\Theta : V \to \mathbb{N}$ is a *priority function* with a finite image that maps each vertex $v \in V$ to a *priority.*

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

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

All other plays are a *tie.*

A strategy for player σ in the 3-valued parity game $\mathcal{G} = (A, \Theta)$ is a function $\zeta : V^* V_\sigma \to V$ such that for all $v \in V_\sigma$ and $v' \in V$: $\zeta(v) = v'$ implies that Player σ can move from v to v'. A play $v_0 v_1 \ldots$ is said to *conform* to ζ if for all $k \in \mathbb{N}$, such that $v_k \in V_\sigma$: $v_{k+1} = \zeta(v_0 \ldots v_k)$. The strategy is called *memoryless* if for all $w, w' \in V^*$, $v \in V_\sigma$: $\zeta(wv) = \zeta(w'v)$. As such, a memoryless strategy ζ can simply be viewed as a function $V_\sigma \to V$.

A strategy ζ for player σ is a *winning strategy* from $V' \subseteq V$ if every play that starts from a vertex in V' and conforms to ζ is won by player σ. It is called a *non-losing* strategy from V' if every play from $v \in V'$ conforming to ζ is either won by player σ or a tie.

The following can easily be obtained as a generalization from the according result for ordinary parity games [31].

Theorem 3.10. *Player σ has a winning, resp. non-losing strategy in a 3-valued parity game $\mathcal{G}$ iff she has a memoryless winning, resp. non-losing strategy.*

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

Because of this theorem, we restrict ourselves to memoryless strategies in the following without mentioning this explicitly every time.

Remark 3.11. The definition of an ordinary (2-valued) parity game is similar to the definition of the 3-valued game, except that its arena is of the form $A = (V_0, V_1, E)$ such that V_0 and V_1 are defined as before and E is a set of edges. Plays and winning conditions are defined similarly to the 3-valued case, except that the eagerness requirement is omitted. Traditionally, E is total, meaning that every vertex has at least one outgoing edge, thus every play is infinite. A play in an ordinary parity game is always winning to some player, i.e. a tie is not possible.

3.3.2 Model Checking Games as Parity Games

Just as the model checking games for the modal μ-calculus can be seen as ordinary parity games [32], the model checking games of the previous section can be transformed into 3-valued parity games.

Let $M = \{S, R^+, R^-, L)$ be a KMTS with state $s_0 \in S$ and let $\varphi_0 \in \mathcal{L}_\mu$. We associate with these a 3-valued parity game as follows.

Arena The vertices of the 3-valued parity game are the configurations of the model checking game, and its edges are applications of the model checking game rules. The set V_0 consists of all configurations in which Player $\exists$ nominally makes a choice together with configurations in which the play terminates and $\forall$ wins. Thus, $\vee$ and $\Diamond$ configurations, the deterministic configurations and the literal configurations $t \vdash l$ where $\neg l \in L(t)$ are all in V_0. Similarly, the set V_1 consists of all configurations in which Player $\forall$ nominally makes a choice together with configurations in which the play terminates and $\exists$ wins. These include the $\wedge$ and $\Box$ configurations, as well as the literal configurations $t \vdash l$ where $l \in L(t)$. The remaining configurations, i.e. the ones of the form $t \vdash l$ with both $l \notin L(t)$ and $\neg l \notin L(t)$ are set to V_{tie}. An edge is a genuine may edge if in its corresponding model checking game rule, the player at hand chooses a transition sR^-t rather than sR^+t. All other game rule applications lead to must edges.

Priority Function Let $Z_1, \dots, Z_n$ be all the variables occurring in φ_0. They are partially ordered by the relation $\leq_{\varphi_0}$. Note that it is possible to assign to each variable a number $\Theta(Z_i)$ such that for all $i, j = 1, \dots, n$:

- $\Theta(Z_i)$ is even iff Z_i is of type ν;
- $\Theta(Z_i) \leq \Theta(Z_j)$ whenever $Z_i \leq_{\varphi_0} Z_j$.

In fact, setting $\Theta(Z_i)$ to the alternation depth of $fp(Z_i) = \eta Z_i.\psi$ (the fixpoint formula of Z_i), possibly plus 1 to assure that the priority is even iff the fixpoint variable η that binds Z_i in $fp(Z_i)$ is ν, ensures that the above requirements are fulfilled. The priorities on the parity game vertices are then assigned as follows:

$$\Theta(s \vdash \psi) \;:=\; \begin{cases} \Theta(Z) & \text{if } \psi = Z \\ 0 & \text{otherwise} \end{cases}$$

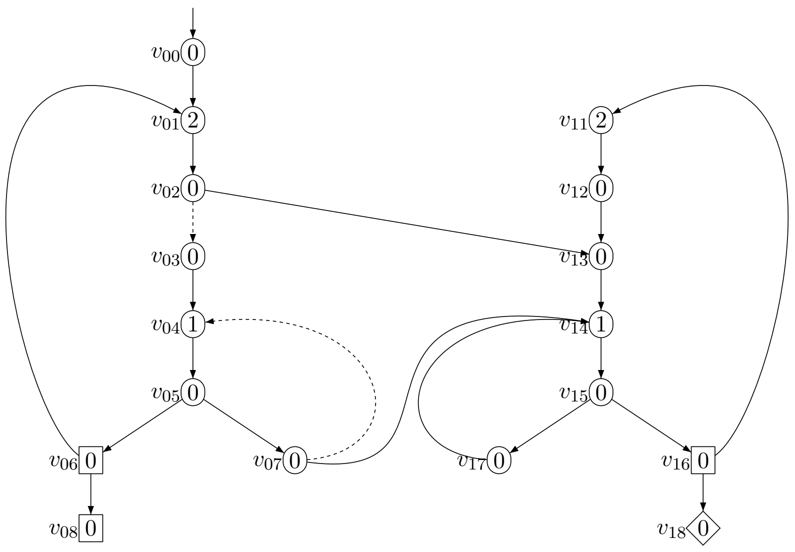


Figure 3.3: The parity game corresponding to the game from Figure 3.2.

Note that Θ has a finite image. Moreover, the maximal priority in the game corresponds to the alternation depth of φ_0.

Example 3.12. In terms of a parity game, the model checking game in Figure 3.2 can be visualized as shown in Figure 3.3. Round vertices denote vertices of Player 0, whereas square vertices are of Player 1. Vertex v_{18} is shaped as a diamond to denote that it is a (dead-end) tie vertex. The numbers labeling the vertices denote their priorities.

The following proposition follows from the fact that the 3-valued parity game is simply a different view onto the model checking game.

Proposition 3.13. *Player $\exists$, resp. $\forall$ has a winning strategy in the model checking game $\Gamma_M(s, \varphi)$ iff Player 0, resp. 1 has a winning strategy in the associated 3-valued parity game from the vertex $s \vdash \varphi$. Moreover, the strategies used in both games are the same.*

Along with the fact that memoryless strategies suffice to determine the result of a 3-valued parity game, this proposition justifies our restriction to memoryless strategies in the setting of model checking games as well.

Remark 3.14. Since the graph of a model checking game need not be total, the corresponding 3-valued parity game might have dead-end vertices. These can be eliminated by applying the following simple transformations.

1. For a dead-end vertex in V_σ (in which V_σ loses), set the priority to $\overline{\sigma}$ and add a must edge back to itself.
2. For a (dead-end) vertex $v \in V_{tie}$, arbitrarily choose $\sigma \in \{0, 1\}$ and add v to V_σ, setting its priority to σ and adding a may edge back to itself.

Note that these changes make the parity game total, i.e. for every $v \in V$ there is a $w \in V$ such that $vE^{-}w$, and in particular there are no vertices in V_{tie}. Moreover, a play looping in the additional edges is won by player σ iff the corresponding play in the original game is won by the same player. In case (1) this is because of the assigned priority; in case (2) this is because the assigned priority which repeats infinitely often is associated with a player who is forced to move along a may edge. Hence, the play is going to be a tie.

Moreover, this construction preserves the set of vertices of the game. It also preserves the player (if any) that has a winning strategy from each vertex, and the same strategies can be used in both games (with the exception that to get a strategy for Player σ in the total game from a strategy in the original game, one has to add to the strategy the self loops that were added to dead-end vertices of Player σ).

3.4 Concluding Remarks

This chapter extends the 2-valued model checking game for the μ-calculus to the 3-valued case, where the model is an abstract KMTS and a third truth value, $\perp$, is possible. The additional truth value corresponds to the new possibility of a *tie* in the game. The 3-valued model checking game is a special case of a 3-valued parity game, also defined in this chapter.

In the following chapters we present algorithms for *solving* 3-valued parity games, i.e., determining the player that has a winning strategy in the game. Due to the correspondence between the model checking game and the model checking problem, this results in a model checking algorithm for the μ-calculus w.r.t. the 3-valued semantics.

The result of this chapter also holds for infinite-state KMTSs. However, in the following chapters, when talking about solving the games, we restrict the discussion to finite KMTSs.

Chapter 4

Game-Based Abstraction-Refinement: Direct Approach

4.1 Introduction

This chapter presents a game-based abstraction-refinement approach for specifications in the μ-calculus, interpreted over a 3-valued semantics. The approach combines a game-based model checking algorithm for abstract models together with an automatic refinement for the case where the model checking result is indefinite.

Model checking is performed by determining the winner, i.e. the player that has a winning strategy, in the corresponding 3-valued model checking game, which we view as a 3-valued parity game (see Chapter 3). Each outcome (winner) corresponds to a truth value. In order to determine the winner of the game, if there is one, we adapt the recursive algorithm for solving parity games by Zielonka [81] using the presentation in [53]. The algorithm in [81] recursively computes the set of configurations in which one of the players has a winning strategy. It then concludes that in all other configurations the other player has a winning strategy.

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

The game-based model checking has the advantage of combining the system model and the checked formula into one structure (the game board). This enables *local* model checking, where only the parts of the model that are relevant to the satisfaction (or falsification) of the checked formula are explored [77]. This combined structure can be computed "on-the-fly", limited to the reachable states of the model, which carries another advantage. Furthermore, the game-based model checking provides auxiliary information that explains its result. Such information can help analyzing the result. For example, in [74] the result is used to produce counterexamples.

The 3-valued semantics, used in our work, allows it to be used for verification as well as falsification. Still, model checking of an abstract model might result in the indefinite result, which calls for refinement. Refinement in this case is aimed at eliminating indefinite results of the model checking, rather than false results.

In case the model checking game results in a tie, meaning the model checking result is indefinite, we identify a cause for the tie and try to eliminate it by refining the abstract model. More specifically, we adapt the presented algorithm to keep track of why a vertex in the game is classified as a tie. We then exploit the information gathered by the algorithm in order to determine a criterion for refinement. Once a criterion for refinement is chosen, the refinement is traditionally done by splitting abstract states. In our case, the refinement is applied only to the parts of the model from which tie is possible. Vertices from which there is a winning strategy for one of the players are not changed. Thus, the refined abstract models do not grow unnecessarily. The result is an incremental abstraction-refinement scheme, where definite results (true or false) from previous iterations are re-used. If the concrete model is finite then our abstraction-refinement is guaranteed to terminate with a definite result.

4.1.1 Related Work

Previous works [57, 65, 66, 60, 6] suggested abstraction-refinement mechanisms for various branching time logics over *2-valued* semantics, for *specific* abstractions. In [57] the authors consider the logic ECTL – the existential fragment of CTL, in addition to the universal fragment ACTL. Yet, their work does not allow a combination of existential and universal quantifiers. The *full* CTL is considered in [66, 60, 6], while [65] considers the full μ-calculus. These works are not restricted to the universal or existential fragment, yet they are designed for *specific* abstractions: in [65, 66] the concrete and abstract systems share the same state space. The simplification is based on taking supersets and subsets of a given set with a more compact BDD representation. In [60] the verified system has to be described as a cartesian product of machines. Finally, in [6] the abstraction collapses all states that satisfy the same subformulas of the checked formula into an abstract state. Our work uses a 3-valued semantics and is applicable to *any* abstraction defined via a set of abstract states and a concretization function.

Model checking of the μ-calculus w.r.t. the 3-valued semantics has been considered for example in [45, 37, 39]. There, a 3-valued μ-calculus model checking problem is reduced to two 2-valued μ-calculus model checking problems. No new model checking algorithm has to be given. However, the underlying transition system has to be studied and therefore be stored twice. Furthermore, these works lack a refinement mechanism, which prevents them from comprising an automatic abstraction-refinement framework. While the reduction to the 2-valued case suffices to decide model checking, it is unclear how to do refinement in a suitable manner. It is the refinement based on the model checking algorithm which rules out the reduction approach taken in these works and justifies our direct approach.

Model checking of multi-valued branching time logics and corresponding tools have been studied in [49, 48, 17]. These papers focus on CTL or CTL* rather than on the more expressive μ-calculus.

The closest work to the work described in this chapter is our previous work [74], where we suggested a game-based framework for abstraction-refinement for CTL w.r.t. a 3-valued semantics. While it is relatively simple to extend this approach to the alternation-free μ-calculus, the extension to the full μ-calculus is not trivial. This is because, in the game board for the alternation-free μ-calculus each strongly connected component can be uniquely identified by a single fixpoint. For the full μ-calculus, this is not the case anymore, thus a more complicated algorithm is needed in order to determine who has the winning strategy.

In a more recent work [35], we have suggested another game-based abstraction-refinement approach based on the 3-valued semantics[1]. There, predicate abstraction is used and the abstract model is implicit since the refinement is applied directly on the game board, rather than on the abstract model. More substantially, in [35], the 3-valued game is solved via a reduction to two 2-valued games. Refinement is based on finding a cause for the tie result, but the determination of a criterion for refinement is separated from the (3-valued) model checking. Namely, it does not consider the run of the model checking algorithm. Instead, three different heuristics for refinement determination are presented. The relevance of the chosen criterion for the tie result is therefore less clear. Furthermore, rather than considering a *global* refinement that splits every abstract state in the indefinite part of the graph in a refinement step, refinement in [35] is applied only locally, at a *single* abstract state, i.e., the lazy abstraction technique [44] for safety properties is adapted to the μ-calculus.

4.2 Model Checking via Solving 3-Valued Parity Games

In this section we develop a game-based 3-valued model checking algorithm. The correspondence between model checking, the model checking game and the 3-valued parity games presented in the previous chapter (see Proposition 3.13 along with Theorem 3.3) implies that the model checking problem for a state s of a KMTS and a formula $\varphi \in \mathcal{L}_\mu$ reduces to determining the player (if any) that has a winning strategy in the 3-valued parity game that corresponds to the model checking game $\Gamma_M(s, \varphi)$. In the remainder of this section we therefore discuss solving 3-valued parity games, which means determining the player (if any) that has a winning strategy from every vertex.

Note that there are three different outcomes for every vertex in a 3-valued parity game: Either Player 0 or Player 1 or none of them has a winning strategy. By the definition of a winning strategy it is obviously not possible for both players to have a winning strategy (in the context of 3-valued model checking games this is implied by Theorem 3.3 as well). Therefore, solving the 3-valued parity game amounts to partitioning its set of vertices into three winning sets: W_0, W_1, W_{tie}, where for $\sigma \in \{0, 1\}$, the set W_σ consists of all the vertices from which Player σ has a winning strategy and the set W_{tie} consists of all the vertices from which none of the players has a winning strategy. We sometimes say that Player σ *wins*, or is the *winner*, in the vertices of W_σ.

When applied to model checking φ in the state s of a KMTS, we check when the algorithm terminates whether the vertex $v = s \vdash \varphi$ is in W_0, W_1, or W_{tie} and conclude that the model checking result is tt, ff or $\perp$, respectively. While the previous chapter was applicable to both finite and infinite KMTSs, we now restrict the discussion to finite KMTSs, where the corresponding 3-valued parity game is finite (i.e., has a finite set of vertices).

[1]In fact, the underlying abstract model in [35] is more expressive and uses must hyper-transitions (see Chapter 6).

4.2.1 Solving 3-Valued Parity Games

Let $\mathcal{G} = (A, \Theta)$ be a (finite) 3-valued parity game with arena $A = (V_0, V_1, V_{tie}, E^+, E^-)$. We adapt to the 3-valued case the recursive algorithm for solving parity games by Zielonka [81] using the presentation in [53]. Its recursive nature makes it easy to understand and analyze, allows simple correctness proofs, and can be used as a basis for refinement (as we will see in Section 4.3).

The main idea of the algorithm presented in [81] is as follows. In each recursive call, σ denotes the parity of the maximal priority in the current game. The algorithm computes the set $W_{\overline{\sigma}}$ iteratively and the remaining vertices form W_σ. In our 3-valued game, we again compute $W_{\overline{\sigma}}$ iteratively, but we then add a phase where we also compute W_{tie} iteratively. Only then, we set W_σ to the remaining vertices.

We start with some definitions. For $X \subseteq V$, the subgraph of $\mathcal{G}$ induced by X, denoted by $\mathcal{G}[X]$, is $(A|_X, \Theta|_X)$ where $A|_X = (V_0 \cap X, V_1 \cap X, V_{tie} \cap X, E^+ \cap X \times X, E^- \cap X \times X)$ and $\Theta|_X$ is the restriction of Θ to X. Note that vertices in V_σ might become dead-ends in $\mathcal{G}[X]$, in which case they are winning for Player $\overline{\sigma}$.

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

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

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

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

$$\begin{array}{lcl}
\text{Attr!}^0_\sigma(\mathcal{G}, X) & = & X \cup D_{\overline{\sigma}} \\
\text{Attr!}^{i+1}_\sigma(\mathcal{G}, X) & = & \text{Attr!}^i_\sigma(\mathcal{G}, X) \\
 & & \cup\ \{v \in V_\sigma \setminus D_\sigma \mid \exists v'.vE^+v' \wedge v' \in \text{Attr!}^i_\sigma(\mathcal{G}, X)\} \\
 & & \cup\ \{v \in V_{\overline{\sigma}} \setminus D_{\overline{\sigma}} \mid \forall v'.vE^-v' \implies v' \in \text{Attr!}^i_\sigma(\mathcal{G}, X)\} \\
\text{Attr!}_\sigma(\mathcal{G}, X) & = & \bigcup\{\text{Attr!}^i_\sigma(\mathcal{G}, X) \mid i \geq 0\} \\
\text{Attr?}^0_\sigma(\mathcal{G}, X) & = & X \cup D_{\overline{\sigma}} \cup D_{tie} \\
\text{Attr?}^{i+1}_\sigma(\mathcal{G}, X) & = & \text{Attr?}^i_\sigma(\mathcal{G}, X) \\
 & & \cup\ \{v \in V_\sigma \setminus D_\sigma \mid \exists v'.vE^-v' \wedge v' \in \text{Attr?}^i_\sigma(\mathcal{G}, X)\} \\
 & & \cup\ \{v \in V_{\overline{\sigma}} \setminus D_{\overline{\sigma}} \mid \forall v'.vE^+v' \implies v' \in \text{Attr?}^i_\sigma(\mathcal{G}, X)\} \\
\text{Attr?}_\sigma(\mathcal{G}, X) & = & \bigcup\{\text{Attr?}^i_\sigma(\mathcal{G}, X) \mid i \geq 0\}
\end{array}$$

The latter definition of the attractor sets provides a method for computing them. As i increases, we calculate $\text{Attr!}^i_\sigma(\mathcal{G}, X)$ or $\text{Attr?}^i_\sigma(\mathcal{G}, X)$ until it is the same as $\text{Attr!}^{i-1}_\sigma(\mathcal{G}, X)$ or $\text{Attr?}^{i-1}_\sigma(\mathcal{G}, X)$, respectively. It is also easy to compute, for each v in $\text{Attr!}_\sigma(\mathcal{G}, X)$ or $\text{Attr?}_\sigma(\mathcal{G}, X)$, the corresponding strategy that allows Player σ to indeed attract the play to the proper vertices as required. For example, if $v \in \mathrm{Attr!}^{i+1}_{\sigma}(\mathcal{G}, X) \setminus \mathrm{Attr!}^{i}_{\sigma}(\mathcal{G}, X)$ then choose $v' \in \mathrm{Attr!}^{i}_{\sigma}(\mathcal{G}, X)$ with $vE^{+}v'$ which must exist by definition.

Note that $\mathrm{Attr!}^{i}_{\sigma}(\mathcal{G}, X) \subseteq \mathrm{Attr?}^{i}_{\sigma}(\mathcal{G}, X)$, and that for $X' = V \setminus \mathrm{Attr?}_{\sigma}(\mathcal{G}, X)$ we have $X' = \mathrm{Attr!}_{\overline{\sigma}}(\mathcal{G}, X')$. Thus, the corresponding must and may attractors partition V. Furthermore:

Lemma 4.1. *For every 3-valued parity game $\mathcal{G}$ and set of vertices X we have*

1. $\mathit{Attr!}_{\sigma}(\mathcal{G}, \mathit{Attr!}_{\sigma}(\mathcal{G}, X)) = \mathit{Attr!}_{\sigma}(\mathcal{G}, X)$.

2. $\mathit{Attr?}_{\sigma}(\mathcal{G}, \mathit{Attr?}_{\sigma}(\mathcal{G}, X)) = \mathit{Attr?}_{\sigma}(\mathcal{G}, X)$.

Solving the Game

We present a recursive algorithm `SolveGame`($\mathcal{G}$) (see Figure 4.4) that computes the sets W_0, W_1, and W_{tie} for a 3-valued parity game $\mathcal{G}$. Let n be the maximum priority occurring in $\mathcal{G}$.

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

Since the maximum priority of $\mathcal{G}$ is 0, Player 1 can only win $\mathcal{G}$ on dead-ends in V_0 or vertices from which he can eagerly attract the play to such a dead-end. This is exactly $\mathrm{Attr!}_1(\mathcal{G}, \emptyset)$, and the strategy for Player 1 follows his strategy for the must attractor set. From the rest of the vertices Player 1 does not have a winning strategy. For vertices in $V \setminus \mathrm{Attr?}_1(\mathcal{G}, \emptyset)$, Player 0 can always avoid reaching dead-ends in $V_0 \cup V_{tie}$, while playing eagerly. Since the maximum priority in this subgraph is 0, it is easy to see that she wins in such vertices. Furthermore, since $W_0 = V \setminus \mathrm{Attr?}_1(\mathcal{G}, \emptyset)$ we know that $W_0 = \mathrm{Attr!}_0(\mathcal{G}, W_0)$ (as noted above these must and may attractors partition V). Thus in order to find the strategy for Player 0 it suffices to compute her strategy for the must attractor set. The remaining vertices in $\mathrm{Attr?}_1(\mathcal{G}, \emptyset) \setminus \mathrm{Attr!}_1(\mathcal{G}, \emptyset)$ are a subset of $\mathrm{Attr?}_1(\mathcal{G}, \emptyset)$, which is why Player 0 does not win from them (and neither does Player 1, as previously claimed). Therefore none of the players wins in $\mathrm{Attr?}_1(\mathcal{G}, \emptyset) \setminus \mathrm{Attr!}_1(\mathcal{G}, \emptyset)$.

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

We first compute $W_{\overline{\sigma}}$ in $\mathcal{G}$. This is done by the algorithm `ComputeOpponentWin` shown in Figure 4.2. The sets used by the algorithm are illustrated by Figure 4.1.

Intuitively, in each iteration we hold a subset of the winning region of Player $\overline{\sigma}$. We first extend it to $X_{\overline{\sigma}}$ by using the must-attractor set of Player $\overline{\sigma}$ (which ensures his eagerness, line 4). From the remaining vertices, we disregard those from which Player σ can attract the play to a vertex with maximum priority n, perhaps by giving up her eagerness. Left are the vertices in Y (line 7) and Player σ is basically trapped in it. She can only "escape" from it to $X_{\overline{\sigma}}$. Thus, we can add the winning region of Player $\overline{\sigma}$ in $\mathcal{G}[Y]$ to his winning region in $\mathcal{G}$. This way, each iteration results in a better (bigger)

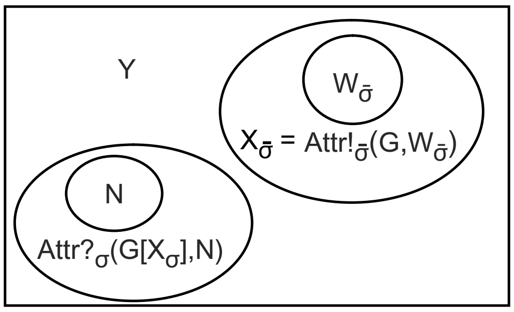

Y
$W_{\bar\sigma}$
$X_{\bar\sigma}$ = Attr!$_{\bar\sigma}$(G,$W_{\bar\sigma}$)
N
Attr?$_\sigma$(G[X_σ],N)

Figure 4.1: Illustration of the sets used by `ComputeOpponentWin`, and of those used by `ComputeNoWin`, in which the must-attractor is replaced by a may-attractor and vice versa.

```
Algorithm ComputeOpponentWin($\mathcal{G}$, $\sigma$, $n$)
1: $W_{\overline{\sigma}} := \emptyset$
2: repeat
3:     $W'_{\overline{\sigma}} := W_{\overline{\sigma}}$
4:     $X_{\overline{\sigma}} :=$ Attr!$_{\overline{\sigma}}(\mathcal{G}, W_{\overline{\sigma}})$
5:     $X_\sigma := V \setminus X_{\overline{\sigma}}$
6:     $N := \{v \in X_\sigma \mid \Theta(v) = n\}$
7:     $Y := X_\sigma \setminus$ Attr?$_\sigma(\mathcal{G}[X_\sigma], N)$
8:     $(Y_0, Y_1, Y_{tie}) :=$ SolveGame($\mathcal{G}[Y]$)
9:     $W_{\overline{\sigma}} := X_{\overline{\sigma}} \cup Y_{\overline{\sigma}}$
10:until $W'_{\overline{\sigma}} = W_{\overline{\sigma}}$
11:return $W_{\overline{\sigma}}$
```

Figure 4.2: Computation of winning vertices for the opponent: `ComputeOpponentWin`.

underapproximation of the winning region of Player $\overline{\sigma}$ in $\mathcal{G}$, until the full region is found (line 10). The proof of the correctness of the algorithm follows.

Lemma 4.2. *For every X_σ as used in the algorithm* `ComputeOpponentWin` *(see Figure 4.2), $\mathcal{G}[X_\sigma]$ is a subgame w.r.t. σ.*

Proof. We show that no vertex, other than vertices in $V_{\overline{\sigma}}$, becomes a dead-end when moving from $\mathcal{G}$ to $\mathcal{G}[X_\sigma]$. Let $v \in X_\sigma$ be a vertex in V_σ which is not a dead-end in $\mathcal{G}$. Suppose to the contrary that all of its successors are in $V \setminus X_\sigma = X_{\overline{\sigma}}$. This means that $v \in \text{Attr!}_{\overline{\sigma}}(\mathcal{G}, X_{\overline{\sigma}})$ and therefore $v \in X_{\overline{\sigma}}$ as well (by Lemma 4.1 since $X_{\overline{\sigma}}$ is a must-attractor), in contradiction. □

Lemma 4.3. *For every Y as used in the algorithm* `ComputeOpponentWin` *(see Figure 4.2), $\mathcal{G}[Y]$ is a subgame.*

Proof. We show that no vertex becomes a dead-end when moving from $\mathcal{G}$ to $\mathcal{G}[Y]$. Let $v \in Y$ be a vertex which is not a dead-end in $\mathcal{G}$.

We first show that v is not a dead-end in $\mathcal{G}[X_\sigma]$. By Lemma 4.2, if v is a dead-end in $\mathcal{G}[X_\sigma]$, then it belongs to $V_{\overline{\sigma}}$ in $\mathcal{G}$, and it became a dead-end (of Player $\overline{\sigma}$) in $\mathcal{G}[X_\sigma]$. Therefore $v \in \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$, in contradiction to $v \in Y = X_\sigma \setminus \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$.

We conclude that v is not a dead-end in $\mathcal{G}[X_\sigma]$, and thus has a successor in X_σ. It remains to show that at least one such successor is not in $\text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$. This is because if all the successors in X_σ are also in $\text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$ (provided that at least one successor exists in X_σ), then $v \in \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N))$, and therefore $v \in \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$ as well (by Lemma 4.1), in contradiction to $v \in Y = X_\sigma \setminus \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$. ☐

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

Lemma 4.4. *At the beginning of each iteration in the algorithm* `ComputeOpponentWin` *(see Figure 4.2), $W_{\overline{\sigma}}$ is a winning region for Player $\overline{\sigma}$ in $\mathcal{G}$.*

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

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

We now show that $Y_{\overline{\sigma}}$ is also a winning region of Player $\overline{\sigma}$ in $\mathcal{G}$. We know that it is a winning region for him in $\mathcal{G}[Y]$ (by the correctness of the algorithm `SolveGame` for games with a maximum priority smaller than n). As for $\mathcal{G}$, for every vertex in $Y_{\overline{\sigma}}$, as long as the play remains in Y, Player $\overline{\sigma}$ can use his strategy for $\mathcal{G}[Y]$. Since $\mathcal{G}[Y]$ is a subgame, Player $\overline{\sigma}$ will always be able to stay within Y in his moves and if the play stays there, then he wins (since he uses his winning strategy). Clearly Player σ cannot move from Y to $X_\sigma \setminus Y = \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$. Otherwise the vertex $v \in Y \subseteq X_\sigma$ where this is done belongs to $\text{Attr?}_\sigma(\mathcal{G}[X_\sigma], \text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N))$ (because the same move is possible in $\mathcal{G}[X_\sigma]$). Hence v belongs to $\text{Attr?}_\sigma(\mathcal{G}[X_\sigma], N)$ as well (by Lemma 4.1), in contradiction to $v \in Y$. Finally, if Player σ moves to $V \setminus X_\sigma = X_{\overline{\sigma}}$, then Player $\overline{\sigma}$ will use his strategy for $X_{\overline{\sigma}}$ in $\mathcal{G}$ and also win.

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

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

Lemma 4.5. *When $W'_{\overline{\sigma}} = W_{\overline{\sigma}}$, then $V \setminus W_{\overline{\sigma}}$ is a non-winning region for Player $\overline{\sigma}$ in $\mathcal{G}$.*

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

Clearly, Player $\overline{\sigma}$ cannot move from X_σ to $X_{\overline{\sigma}}$ without compromising his eagerness. Otherwise the vertex $v \in X_\sigma$ where this is done belongs to $\mathrm{Attr!}_{\overline{\sigma}}(\mathcal{G}, X_{\overline{\sigma}})$ and therefore to $X_{\overline{\sigma}}$ as well (by Lemma 4.1 since $X_{\overline{\sigma}}$ is a must-attractor). This contradicts $v \in X_\sigma$. Hence, Player $\overline{\sigma}$ cannot win by moving to $X_{\overline{\sigma}}$ and since $\mathcal{G}[X_\sigma]$ is a subgame w.r.t. σ, Player σ is never obliged to move to $X_{\overline{\sigma}}$.

Consider the case where the play stays in X_σ. In order to prevent Player $\overline{\sigma}$ from winning, Player σ will play as follows. If the current configuration is in Y, then Player σ will use her strategy on $\mathcal{G}[Y]$ for preventing Player $\overline{\sigma}$ from winning (such a strategy exists since $Y_{\overline{\sigma}} = \emptyset$). If the play visits a vertex $v \in N$ which is controlled by Player σ, then Player σ will move to any successor v' inside X_σ. Such a successor must exist since vertices in N are never dead-ends in $\mathcal{G}$. Furthermore, the vertex at hand belongs to V_σ, thus since $\mathcal{G}[X_\sigma]$ is a subgame w.r.t. σ (by Lemma 4.2), it remains non dead-end in $\mathcal{G}[X_\sigma]$. If the play visits $\mathrm{Attr?}_\sigma(\mathcal{G}[X_\sigma], N) \setminus N$, then Player σ will use his strategy to either cause Player $\overline{\sigma}$ to give up eagerness, or to attract the play in a finite number of steps to N or $D'_{\overline{\sigma}} \cup D_{tie}$ (such a strategy exists by the definition of a may-attractor set). We use $D'_{\overline{\sigma}}$ to denote the dead-end vertices of Player $\overline{\sigma}$ in $\mathcal{G}[X_\sigma]$. Since $\mathcal{G}[X_\sigma]$ is not necessarily a subgame w.r.t. $\overline{\sigma}$, $D'_{\overline{\sigma}}$ may contain non dead-end vertices of Player $\overline{\sigma}$ from $\mathcal{G}$ that became dead-ends in $\mathcal{G}[X_\sigma]$. However, this means that all their successors are in $X_{\overline{\sigma}}$, and as stated before Player $\overline{\sigma}$ cannot move eagerly from X_σ to $X_{\overline{\sigma}}$, thus he cannot win in them in $\mathcal{G}$ as well.

This strategy indeed prevents Player $\overline{\sigma}$ from winning: Three cases can occur. First, from some moment on, the play stays forever inside of Y. In this case Player $\overline{\sigma}$ cannot win (by the correctness of the algorithm `SolveGame` in $\mathcal{G}[Y]$ since $Y_{\overline{\sigma}} = \emptyset$). Second, Player $\overline{\sigma}$ is not eageror the play reaches a dead-end in $(D_{\overline{\sigma}} \cup D_{tie}) \cap (\mathrm{Attr?}_\sigma(\mathcal{G}[X_\sigma], N) \setminus N)$, in which case Player $\overline{\sigma}$ does not win as well. Third, the play visits infinitely often the maximum priority n (in the set N) and Player $\overline{\sigma}$ cannot win. ☐

Corollary 4.6. *The result of* `ComputeOpponentWin` *is the full winning region of Player* $\overline{\sigma}$ *in* $\mathcal{G}$.

In the original algorithm in [81], given the set $W_{\overline{\sigma}}$, we could conclude that all the remaining vertices form the winning region of Player σ in $\mathcal{G}$. Yet, this is not the case here.

Given the set $W_{\overline{\sigma}}$, we now divide the remaining vertices into W_{tie} and W_σ. To do so, we first compute the set *nowin* of vertices in $\mathcal{G}$ from which Player σ does not have a winning strategy, i.e. Player $\overline{\sigma}$ has a strategy that prevents Player σ from winning. This is again done iteratively, by the algorithm `ComputeNoWin`, given in Figure 4.3. Figure 4.1, which illustrates the sets used by `ComputeOpponentWin` also illustrates the sets used by Figure 4.1, when replacing the must-attractor by a may-attractor and vice versa.

The algorithm `ComputeNoWin` resembles the algorithm `ComputeOpponentWin`. The initialization here is to $W_{\overline{\sigma}}$, since this is clearly a non-winning region of Player σ. Furthermore, in this case after the recursive call to `SolveGame`$(\mathcal{G}[Y])$, the set $X_{\overline{\sigma}}$ is extended not only by the winning region of Player $\overline{\sigma}$ in $\mathcal{G}[Y]$, $Y_{\overline{\sigma}}$, but also by the tie-region Y_{tie} (line 9). Apart from those differences, one can see that the only difference is that the use of a must-attractor set is replaced by a may-attractor set and vice versa. This is because in the case of `ComputeOpponentWin` we are after a definite win of Player $\overline{\sigma}$,

```
Algorithm ComputeNoWin ($\mathcal{G}$, $\sigma$, $n$, $W_{\overline{\sigma}}$)
1: $nowin := W_{\overline{\sigma}}$
2: repeat
3:     $nowin' := nowin$
4:     $X_{\overline{\sigma}} :=$ Attr?$_{\overline{\sigma}}(\mathcal{G}, nowin)$
5:     $X_\sigma := V \setminus X_{\overline{\sigma}}$
6:     $N := \{v \in X_\sigma \mid \Theta(v) = n\}$
7:     $Y := X_\sigma \setminus$ Attr!$_\sigma(\mathcal{G}[X_\sigma], N)$
8:     $(Y_0, Y_1, Y_{tie}) :=$ SolveGame($\mathcal{G}[Y]$)
9:     $nowin := X_{\overline{\sigma}} \cup Y_{\overline{\sigma}} \cup Y_{tie}$
10:until $nowin' = nowin$
11:return $nowin$
```

Figure 4.3: Computation of vertices in which no win is possible: `ComputeNoWin`.

whereas in the case of `ComputeNoWin` we also allow a tie, therefore may edges take a different role. Namely, in this case, when we extend the current set *nowin* (line 4), we use a may-attractor set of Player $\overline{\sigma}$ because when our goal is to prevent Player σ from winning, we allow Player $\overline{\sigma}$ to not be eager. On the other hand, in the computation of Y we now remove from $X_{\overline{\sigma}}$ only the vertices from which Player σ can *eagerly* attract the play to the maximum priority (using the must-attractor set, line 7). This is because only such vertices cannot contribute to the goal of preventing Player σ from winning. Other vertices where he can reach the maximum priority, but only at the expense of eagerness, can still be of use for this goal.

Lemma 4.7. *For every X_σ as used in the algorithm* `ComputeNoWin` *(see Figure 4.3), $\mathcal{G}[X_\sigma]$ is a subgame.*

Proof. We show that no vertex becomes a dead-end when moving from $\mathcal{G}$ to $\mathcal{G}[X_\sigma]$. Let $v \in X_\sigma$ be a vertex which is not a dead-end in $\mathcal{G}$. Suppose to the contrary that all its successors are in $V \setminus X_\sigma = X_{\overline{\sigma}}$ (where we know that at least one successor exists). This means that $v \in \text{Attr?}_{\overline{\sigma}}(\mathcal{G}, X_{\overline{\sigma}})$ and therefore $v \in X_{\overline{\sigma}}$ as well (by Lemma 4.1 since $X_{\overline{\sigma}}$ is a may-attractor), in contradiction. ☐

Lemma 4.8. *For every Y as used in the algorithm* `ComputeNoWin` *(see Figure 4.3), $\mathcal{G}[Y]$ is a subgame.*

Proof. We show that no vertex becomes a dead-end when moving from $\mathcal{G}$ to $\mathcal{G}[Y]$. Let $v \in Y$ be a vertex which is not a dead-end in $\mathcal{G}$. Then, by Lemma 4.7, v has a successor in X_σ. Furthermore, a simple proof by contradiction, similar to the proof of Lemma 4.7, shows that if $v \in V_\sigma$, then it has a must successor in X_σ. It remains to show that at least one such successor is not in $\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$. This is because if all the successors in X_σ, including the must successor for $v \in V_\sigma$, are also in $\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$, then $v \in \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N))$, and therefore $v \in \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$ as well (by Lemma 4.1), in contradiction to $v \in Y = X_\sigma \setminus \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$. ☐

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

Lemma 4.9. *At the beginning of each iteration, the set nowin is a non-winning region for Player σ in $\mathcal{G}$.*

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

Clearly, $X_{\overline{\sigma}} = \text{Attr?}_{\overline{\sigma}}(\mathcal{G}, nowin)$ is also a non-winning region for Player σ in $\mathcal{G}$: Player $\overline{\sigma}$ can use his strategy to either (1) cause Player σ to not be eager, (2) attract the play to a dead-end in $D_\sigma \cup D_{tie}$ (in which case Player σ cannot win), or (3) attract the play to $nowin$, where he can use his strategy for preventing Player σ from winning in $\mathcal{G}$ (by the induction hypothesis).

We now show that $Y_{\overline{\sigma}} \cup Y_{tie}$ is also a non-winning region of Player σ in $\mathcal{G}$. We know that it is a non-winning region for her in $\mathcal{G}[Y]$ (by the correctness of the algorithm `SolveGame` for games with a maximum priority smaller than n). As for $\mathcal{G}$, for every vertex in $Y_{\overline{\sigma}} \cup Y_{tie}$, as long as the play remains in Y, Player $\overline{\sigma}$ can use his strategy in $\mathcal{G}[Y]$. Since this is a subgame, Player $\overline{\sigma}$ will always be able to remain in Y in his moves and if the play stays there Player σ will not win. Clearly Player σ cannot eagerly move from Y to $X_\sigma \setminus Y = \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$. Otherwise the vertex $v \in Y \subseteq X_\sigma$ where this is done belongs to $\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], \text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N))$ (because the same move is possible in $\mathcal{G}[X_\sigma]$). Hence v belongs to $\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$ as well (by Lemma 4.1), in contradiction to $v \in Y$. Finally, if Player σ moves to $V \setminus X_\sigma = X_{\overline{\sigma}}$, then Player $\overline{\sigma}$ will use his strategy for $X_{\overline{\sigma}}$ in $\mathcal{G}$ to prevent her from winning.

We conclude that $X_{\overline{\sigma}} \cup Y_{\overline{\sigma}} \cup Y_{tie}$ is a non-winning region for Player σ in $\mathcal{G}$. ☐

This lemma ensures that the final result $nowin$ of `ComputeNoWin` is indeed a subset of the non-winning region of Player σ in $\mathcal{G}$. It remains to show that this is actually an equality, i.e. that no non-winning vertices are missing.

Lemma 4.10. *When $nowin' = nowin$, then $V \setminus nowin$ is a winning region for Player σ in $\mathcal{G}$.*

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

Clearly, Player $\overline{\sigma}$ cannot move from X_σ to $X_{\overline{\sigma}}$. Otherwise the vertex $v \in X_\sigma$ where this is done belongs to $\text{Attr?}_{\overline{\sigma}}(\mathcal{G}, X_{\overline{\sigma}})$ and therefore to $X_{\overline{\sigma}}$ as well (by Lemma 4.1 since $X_{\overline{\sigma}}$ is a may-attractor). This contradicts $v \in X_\sigma$. Hence, Player $\overline{\sigma}$ is "trapped" in X_σ and since $\mathcal{G}[X_\sigma]$ is a subgame, Player σ is never obliged to move to $X_{\overline{\sigma}}$.

Consider the case where the play stays in X_σ. In order to win, Player σ will play as follows. If the current configuration is in Y, then Player σ will use his winning strategy on $\mathcal{G}[Y]$ (such a strategy exists since $Y_{\overline{\sigma}} = Y_{tie} = \emptyset$ and $Y_\sigma = Y$). If the play visits a vertex $v \in N$, then Player σ will move to a must successor v' inside X_σ. Such a successor exists because otherwise $v \in \text{Attr?}_{\overline{\sigma}}(\mathcal{G}, X_{\overline{\sigma}})$ and hence also in $X_{\overline{\sigma}}$ (by Lemma 4.1 since $X_{\overline{\sigma}}$ is a

```
Algorithm SolveGame ($\mathcal{G}$)
1: if $V = \emptyset$ then return ($\emptyset$, $\emptyset$, $\emptyset$)
2: $n := max\{\Theta(v) \mid v \in V\}$
3: if $n = 0$ then // return ($W_0$, $W_1$, $W_{tie}$)
4:    return ($V \setminus$ Attr?$_1(\mathcal{G},\emptyset)$, Attr!$_1(\mathcal{G},\emptyset)$, Attr?$_1(\mathcal{G},\emptyset) \setminus$ Attr!$_1(\mathcal{G},\emptyset)$)
5: else
6:    $\sigma := n \mod 2$
7:    $W_{\overline{\sigma}} :=$ ComputeOpponentWin($\mathcal{G}$, $\sigma$, $n$)
8:    $W_\sigma := V \setminus$ ComputeNoWin($\mathcal{G}$, $\sigma$, $n$, $W_{\overline{\sigma}}$)
9:    $W_{tie} := V \setminus (W_{\overline{\sigma}} \cup W_\sigma)$
10:   return ($W_0$, $W_1$, $W_{tie}$)
```

Figure 4.4: Algorithm `SolveGame`.

may-attractor), in contradiction to $v \in N \subseteq X_\sigma$. If the play visits $\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N) \setminus N$, then Player σ will attract it in a finite number of steps to N or $D_{\overline{\sigma}}$, while being eager.

This strategy ensures that Player σ is eagerand is indeed winning: Three cases can occur. First, from some moment on, the play stays forever inside of Y. In this case Player σ wins (by the correctness of the algorithm `SolveGame` in $\mathcal{G}[Y]$). Second, the play reaches a dead-end in $D_{\overline{\sigma}} \cap (\text{Attr!}_\sigma(\mathcal{G}[X_\sigma], N) \setminus N)$, in which case Player σ wins as well. Third, the play visits infinitely often the maximum priority n (in the set N) and Player σ wins. □

Corollary 4.11. `ComputeNoWin` *returns the full non-winning region of Player σ in $\mathcal{G}$.*

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

To sum up, solving the game is achieved by the algorithm `SolveGame` shown in Figure 4.4.

We have suggested an algorithm for computing the winning (and non-winning) regions of the players. In the correctness proofs, we have also defined strategies for the players. The algorithm can also be used for checking a concrete system in which all may edges are also must edges and $V_{tie} = \emptyset$.

Remark 4.12. *Let $\mathcal{G}$ be a 3-valued parity game in which $V_{tie} = \emptyset$ and all edges are must edges. Then W_{tie} computed by the algorithm `SolveGame` is empty.*

Complexity Let l and m denote the number of vertices and edges of $\mathcal{G}$, and, let n be the maximum priority. Computing attractor sets is a reachability problem and as such linear in $m + l = O(m)$. Let $SG(m, l, n)$, $COW(m, l, n)$, and $CNW(m, l, n)$ denote the complexity of `SolveGame`, `ComputeOpponentWin`, and `ComputeNoWin`, respectively. We get the following relations:

$$\begin{aligned} SG(m,l,0) &\leq c_1 \cdot m \\ SG(m,l,n+1) &\leq c_2 \cdot m + COW(l,m,n+1) + CNW(l,m,n+1) \\ COW(m,l,n+1) &\leq l_1 \cdot (c_3 \cdot m + SG(m,l,n)) \\ CNW(m,l,n+1) &\leq l_2 \cdot (c_4 \cdot m + SG(m,l,n)) \end{aligned}$$

where in each recursive call of `solveGame`, $l_1 + l_2 \leq l$. From this we easily conclude that the time complexity of the algorithm is in $O(m \cdot l^n)$. Thus, the complexity is comparable to known upper bounds for solving ordinary (2-valued) parity games.

We sum up this section:

Theorem 4.13. *Algorithm* `SolveGame` *computes the winning regions* (W_0, W_1, W_{tie}) *for a given 3-valued parity game in time exponential in the maximal priority. Furthermore, it can be used to determine the winning strategy for the corresponding winner.*

We conclude that when applied to a model checking game $\Gamma_M(s,\varphi)$, the complexity of `SolveGame` is exponential in the alternation depth of φ.

Example 4.14. We illustrate the algorithm `SolveGame` on the 3-valued parity game $\mathcal{G}$ depicted in Figure 3.3.

```
SolveGame($\mathcal{G}$):   $n = 2$,  $\sigma = 0$        % $D_0 = \emptyset$, $D_1 = \{v_{08}\}$, $D_{tie} = \{v_{18}\}$
  $W_1 :=$ ComputeOpponentWin($\mathcal{G}$, 0, 2) $= \emptyset$        % see below.
  $W_0 := V\setminus$ ComputeNoWin ($\mathcal{G}$, 0, 2, $\emptyset$) $= V \setminus (V \setminus \{v_{08}\}) = \{v_{08}\}$   % see below.
  $W_{tie} := V \setminus (W_0 \cup W_1) = V \setminus \{v_{08}\}$
  SolveGame($\mathcal{G}$) returns ($\{v_{08}\}$, $\emptyset$, $V \setminus \{v_{08}\}$)

ComputeOpponentWin($\mathcal{G}$, $\sigma = 0$, $n = 2$):
  Initially, $W_1 := \emptyset$
  First iteration:
    $X_1 :=$ Attr!$_1(\mathcal{G}, \emptyset) = \emptyset$        % From $X_1$, Player 1 wins by attracting each play to $D_0$.
    $X_0 := V \setminus X_1 = V$        % In $X_0$, the winner is to be determined.
    $N := \{v \in X_0 \mid \Theta(v) = 2\} = \{v_{01}, v_{11}\}$
        % $N$ comprises of vertices with maximal priority.  Attr?$_0(\mathcal{G}[X_0], N) = V$.
    $Y := X_0 \setminus$ Attr?$_0(\mathcal{G}[X_0], N) = V \setminus V = \emptyset$
        % From $Y$, Player 0 cannot escape to the maximal priority.
    $(Y_0, Y_1, Y_{tie}) :=$ SolveGame($\mathcal{G}[Y]$) $= (\emptyset, \emptyset, \emptyset)$
    $W_1 := X_1 \cup Y_1 = \emptyset$        % No vertices were added to $W_1$, thus the loop terminates.
  ComputeOpponentWin($\mathcal{G}$, 0, 2) returns $\emptyset$.

ComputeNoWin ($\mathcal{G}$, $\sigma = 0$, $n = 2$, $W_1 = \emptyset$):
  Initially, $nowin := \emptyset$        % initialized to $W_1$.
  First iteration:
    $X_1 :=$ Attr?$_1(\mathcal{G}, \emptyset) = \{v_{16}, v_{18}\}$    % In $X_1$, either Player 1 wins or a tie occurs.
    $X_0 := V \setminus X_1 = V \setminus \{v_{16}, v_{18}\}$    % In $X_0$, whether Player 0 can win is to be determined.
    $N := \{v \in X_0 \mid \Theta(v) = 2\} = \{v_{01}, v_{11}\}$
        % Attr!$_0(\mathcal{G}[X_0], N) = \{v_{00}, v_{01}, v_{03}, v_{04}, v_{05}, v_{06}, v_{08}, v_{11}\}$.
    $Y := X_0 \setminus$ Attr!$_0(\mathcal{G}[X_0], N) = \{v_{12}, v_{13}, v_{14}, v_{15}, v_{17}, v_{02}, v_{07}\}$
        % From $Y$, Player 0 cannot *eagerly* escape to the maximal priority.
    $(Y_0, Y_1, Y_{tie}) :=$ SolveGame($\mathcal{G}[Y]$) = ($\emptyset$, $Y$, $\emptyset$)    % Recursive call. See below.
    $nowin := X_1 \cup Y_1 \cup Y_{tie} = \{v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{17}, v_{18}, v_{02}, v_{07}\}$
  Second iteration:
    $X_1 :=$ Attr?$_1(\mathcal{G}, nowin) = V \setminus \{v_{08}\}$
    $X_0 := V \setminus X_1 = \{v_{08}\}$
    $N := \{v \in X_0 \mid \Theta(v) = 2\} = \emptyset$    % Attr!$_0(\mathcal{G}[X_0], N) = \{v_{08}\}$
    $Y := X_0 \setminus$ Attr!$_0(\mathcal{G}[X_0], N) = \{v_{08}\} \setminus \{v_{08}\} = \emptyset$
    $(Y_0, Y_1, Y_{tie}) :=$ SolveGame($\mathcal{G}[Y]$) $= (\emptyset, \emptyset, \emptyset)$
    $nowin := X_1 \cup Y_{\overline{\sigma}} \cup Y_{tie} = V \setminus \{v_{08}\}$
  Third iteration:
    $X_1 :=$ Attr?$_1(\mathcal{G}, nowin) = V \setminus \{v_{08}\}$
        % The rest of the execution is the same as the second iteration. *nowin* remains unchanged and
    the loop terminates.
  ComputeNoWin ($\mathcal{G}$, 0, 2, $\emptyset$) returns $V \setminus \{v_{08}\}$
```

We conclude that $W_0 = \{v_{08}\}$, $W_1 = \emptyset$, and that all remaining vertices form the set W_{tie}. We now display the execution of the recursive call SolveGame($\mathcal{G}[Y]$), where $Y = \{v_{12}, v_{13}, v_{14}, v_{15}, v_{17}, v_{02}, v_{07}\}$. The subgame $\mathcal{G}[Y]$ is illustrated in Figure 4.5. In the following, we use primed versions of the set names in order to prevent confusion with the sets used by SolveGame($\mathcal{G}$), in particular Y.

```
SolveGame($\mathcal{G}[Y]$):  $n = 1$,  $\sigma = 1$    % $D'_0 = \emptyset$, $D'_1 = \emptyset$, $D'_{tie} = \emptyset$
  $W'_0 :=$ ComputeOpponentWin($\mathcal{G}[Y]$, 1, 1) $= \emptyset$    % see below.
  $W'_1 := Y \setminus$ ComputeNoWin ($\mathcal{G}[Y]$, 1, 1, $\emptyset$) $= Y \setminus \emptyset = Y$    % see below.
  $W'_{tie} := Y \setminus (W'_0 \cup W'_1) = \emptyset$
  SolveGame($\mathcal{G}[Y]$) returns ($\emptyset$, $Y$, $\emptyset$)

ComputeOpponentWin($\mathcal{G}[Y]$, $\sigma = 1$, $n = 1$):
  Initially, $W'_0 = \emptyset$
  $X'_0 :=$ Attr!$_0(\mathcal{G}[Y], \emptyset) = \emptyset$
  $X'_1 := Y \setminus X'_0 = Y$
  $N := \{v \in X'_1 \mid \Theta(v) = 1\} = \{v_{14}\}$    % Attr?$_1(\mathcal{G}[X'_1], N) = Y$
  $Y' := X'_1 \setminus$ Attr?$_1(\mathcal{G}[X'_1], N) = Y \setminus Y = \emptyset$
  $(Y'_0, Y'_1, Y'_{tie}) :=$ SolveGame($\mathcal{G}[Y']$) $= (\emptyset, \emptyset, \emptyset)$
  $W'_0 := X'_0 \cup Y'_0 = \emptyset$    % No vertices were added to $W'_0$, thus the loop terminates.
```

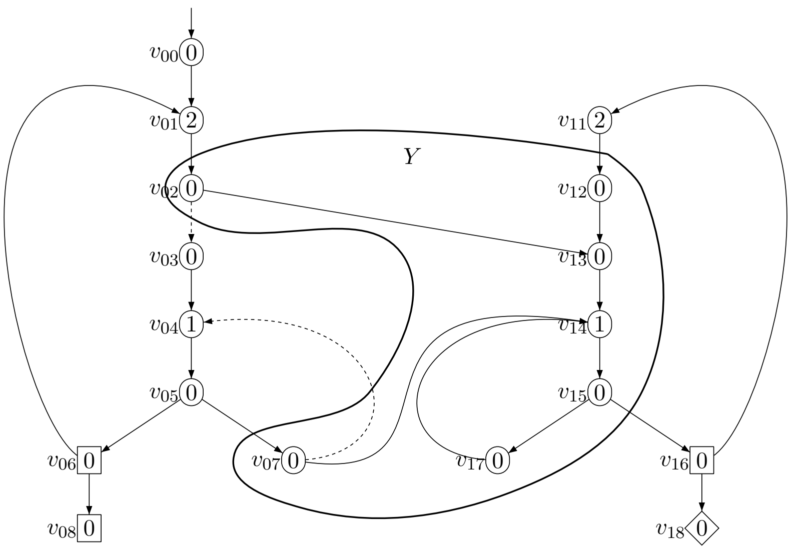


Figure 4.5: Subgame $\mathcal{G}[Y]$ used by `SolveGame` $(\mathcal{G})$ in Example 4.14.

```
    ComputeOpponentWin($\mathcal{G}[Y]$, 1, 1) returns $\emptyset$

  ComputeNoWin ($\mathcal{G}[Y]$, $\sigma = 1$, $n = 1$, $W'_0 = \emptyset$):
    Initially, $nowin' := \emptyset$
    $X'_0 :=$ Attr?$_0(\mathcal{G}[Y], \emptyset) = \emptyset$
    $X'_1 := Y \setminus X'_0 = Y$
    $N := \{v \in X'_1 \mid \Theta(v) = 1\} = \{v_{14}\}$        % Attr!$_1(\mathcal{G}[X'_1], N) = Y$
    $Y' := X'_1 \setminus$ Attr!$_1(\mathcal{G}[X'_1], N) = Y \setminus Y = \emptyset$
    $(Y'_0, Y'_1, Y'_{tie}) :=$ SolveGame($\mathcal{G}[Y']$) $= (\emptyset, \emptyset, \emptyset)$
    $nowin' = X'_0 \cup Y'_0 \cup Y'_{tie} = \emptyset$        % No vertices were added to $nowin'$, thus the loop terminates.
    ComputeNoWin($\mathcal{G}[Y]$, 1, 1, $\emptyset$) returns $\emptyset$
```

Based on Proposition 3.13 and Theorem 3.3 from the previous chapter, the algorithm for solving a 3-valued parity game provides a 3-valued model checking algorithm for KMTSs.

Remark 4.15. The same algorithm can also be used to model check a formula $\varphi \in \mathcal{L}_\mu$ on a KMTS that has a set of initial states S^0. In this case, to determine the value of φ on the KMTS, we need to consider all the vertices in $S^0 \times \{\varphi\}$. If all of them are in W_0, then the value of φ in all the initial states is tt. Thus, the KMTS satisfies φ. If at least one of them is in W_1, then the value of φ in the corresponding initial state is ff, and the KMTS falsifies φ. Otherwise, the value of φ in the KMTS is indefinite. In order to make sure that all the vertices in $S^0 \times \{\varphi\}$ are considered when solving the corresponding 3-valued game, it is possible to add to the 3-valued parity game an auxiliary initial vertex, connected by edges to all the vertices in $S^0 \times \{\varphi\}$.

4.3 Refinement of 3-Valued Parity Games

Assume we are interested in knowing whether a concrete state s_c, represented by an abstract state s_a, satisfies a given formula φ. Let (W_0, W_1, W_{tie}) be the winning sets computed for the 3-valued parity game obtained by the model checking game $\Gamma_M(s_a, \varphi)$ for s_a and φ. By Proposition 3.13 along with Corollary 3.4 if the vertex $v = s_a \vdash \varphi$ is in W_0 or W_1 then the answer to the model checking problem is clear: if $v \in W_0$ then s_c satisfies φ, and similarly, if $v \in W_1$ then s_c falsifies φ. However, if $v \in W_{tie}$, the result is indefinite and we have to refine the abstraction to get the answer.

When considering a KMTS with a set of initial states S^0_A, refinement is needed when at least one of the vertices in $S^0_A \times \{\varphi\}$ is not in W_0 (thus the result is not tt), and none of them is in W_1 (thus the result is also not ff). This means that at least one of the initial vertices is in W_{tie}.

Refinement is performed by splitting the abstract states. As in most cases, our refinement consists of two parts. First, we choose a criterion that tells us how to split the abstract states. We then construct the refined abstract model, using the refined abstract state space. In the rest of this section we focus on the first part, which is also performed in two steps: First, some failure cause is identified. Then, the criterion for altering the abstraction, and splitting all the abstract states, is determined based on a failure analysis.

4.3.1 Identifying a Failure Cause

Given that $v \in W_{tie}$, our goal in the refinement is to find and eliminate at least one of the causes of the indefinite result. Thus, the criterion for splitting the abstract states is obtained from a *failure vertex.* This is a vertex $v' = s'_a \vdash \varphi'$ such that (1) $v' \in W_{tie}$; (2) the classification of v' to W_{tie} *affects* the indefinite result of v; and (3) the indefinite classification of v' can be *changed* by splitting it. The latter requirement means that the vertex v' *itself* is responsible for introducing (some) uncertainty. The other requirements demand that this uncertainty is relevant to the result in v.

The game solving algorithm is adapted to remember for each vertex in W_{tie} a failure vertex, and a *failure cause.* The state s'_a of the failure vertex $v' = s'_a \vdash \varphi'$ is also referred to as the *failure state.* We distinguish between the case where $n = 0$ and the case where $n \geq 1$ in `SolveGame`.

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

Since a similar computation is used in the case $n \geq 1$ as well, we describe it more generally. Namely, we set $\sigma = 0$, meaning that 0 is replaced by σ, 1 is replaced by $\overline{\sigma}$

and the role of W_1 is replaced by $W_{\overline{\sigma}}$. The enriched computation of $\mathrm{Attr?}_{\overline{\sigma}}(\mathcal{G}, \emptyset)$ is as follows.

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

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

o1. If $v' \in V_{\overline{\sigma}}$ and there exists a may edge $v'E^-v''$ such that $v'' \in W_{\overline{\sigma}}$, then v' is a failure vertex, with this edge being the cause [failP$\overline{\sigma}$].

o2. If $v' \in V_\sigma$ and has a may edge $v'E^-v''$ such that $v'' \notin \mathrm{Attr?}^i_{\overline{\sigma}}(\mathcal{G}, \emptyset)$, then v' is a failure vertex, with this edge being the cause [failPσ].

o3. Otherwise, there exists a may (possibly must) edge $v'E^-v''$ such that $v'' \in \mathrm{Attr?}^i_{\overline{\sigma}}(\mathcal{G}, \emptyset) \setminus W_{\overline{\sigma}}$. The failure vertex and cause of v' are those of v''.

Note that the order of the "if" statements in the algorithm determines the failure vertex returned by the algorithm. Different heuristics can be applied regarding their order.

Lemma 4.16. *The computation of failure vertices for $n = 0$ is well defined, meaning that all the possible cases are handled. Furthermore, if the failure cause computed by it is a may edge, then this edge is* not *a must edge.*

Proof. To be convinced of this, one needs to notice that all the possible cases are handled. Moreover, we need to ensure that whenever cases o1 and o2 apply, then the edge chosen as a failure cause is not a must edge. By the definition of $\mathrm{Attr?}_1(\mathcal{G}, \emptyset)$, every vertex $v' \notin W_1$ that is added to the set $\mathrm{Attr?}^{i+1}_1(\mathcal{G}, \emptyset)$ fulfills one of the following possibilities.

- $v' \in V_1$ and has a may edge to a vertex $v'' \in \mathrm{Attr?}^i_1(\mathcal{G}, \emptyset)$. If there exists such an edge for which $v'' \in W_1$, then case o1 applies. In this case we are guaranteed that $(v', v'') \notin E^+$, since otherwise v' would be in W_1 as well. If there is no such edge for which $v'' \in W_1$, then there exists an edge to a vertex $v'' \in \mathrm{Attr?}^i_1(\mathcal{G}, \emptyset) \setminus W_1$ and case o3 applies.

- $v' \in V_0$ and all its must edges are to vertices in $\mathrm{Attr?}^i_1(\mathcal{G}, \emptyset)$. If it has a may edge $v'E^-v''$ such that $v'' \notin \mathrm{Attr?}^i_1(\mathcal{G}, \emptyset)$, then case o2 applies. Furthermore, this edge is not a must edge, because otherwise the condition for adding v' to the attractor set would not be fulfilled. If there is no such edge, then all the outgoing edges of v' are to $\mathrm{Attr?}^i_1(\mathcal{G}, \emptyset)$. If all of them are in W_1, then v' also has to be in W_1, in contradiction. Thus, there exists at least one outgoing edge to $\mathrm{Attr?}^i_1(\mathcal{G}, \emptyset) \setminus W_1$ and case o3 applies.

□

Intuitively, during each iteration of the computation, if the vertex $v' \in W_{tie}$ that is added to $\mathrm{Attr?}^{i+1}_1(\mathcal{G}, \emptyset)$ is not responsible for introducing its indefinite result (cases o1 and o2), then the computation greedily continues with a vertex in W_{tie} that *affects* its indefinite classification (case o3).

One can see that there are three possibilities where we say that the vertex itself is responsible for the uncertainty and consider it a failure vertex: failDE, failP$\overline{\sigma}$ and failPσ. For a vertex in D_{tie} (case failDE), the failure cause is clear. Consider case failP$\overline{\sigma}$. In this case $v' \in V_{\overline{\sigma}}$ is considered a failure vertex, with the may edge to $v'' \in W_{\overline{\sigma}}$ being the failure cause. By Lemma 4.16 we have that this edge is *not* a must edge. The intuition for v' being a failure vertex is that if this edge was a must edge, then it would change the classification of v' to $W_{\overline{\sigma}}$. If no such edge existed, then v' would not be added to $\text{Attr?}^{i+1}_{\overline{\sigma}}(\mathcal{G}, \emptyset)$ and thus to W_{tie}. Finally, consider case failPσ. In this case $v' \in V_\sigma$ has a may edge to v'' which is still unclassified at the time v' is added to $\text{Attr?}_{\overline{\sigma}}(\mathcal{G}, \emptyset)$. This edge is considered a failure cause because if it was a must edge rather than a may edge (which by Lemma 4.16 it is not), then v' would remain unclassified as well for at least one more iteration. Thus it would have a better chance to eventually remain outside the set $\text{Attr?}^{i}_{\overline{\sigma}}(\mathcal{G}, \emptyset)$ until fixpoint is reached, changing the classification of v' to W_σ.

$\boldsymbol{n \geq 1}$**:** We now refer to the case where $n \geq 1$ in `SolveGame`. In this case the set W_{tie} is computed by $V \setminus (W_{\overline{\sigma}} \cup W_\sigma)$. Recall that W_σ is computed by $V\setminus$ `ComputeNoWin`($\mathcal{G}$, σ, n, $W_{\overline{\sigma}}$). Therefore, W_{tie} is equal to `ComputeNoWin`($\mathcal{G}$, σ, n, $W_{\overline{\sigma}}$) $\setminus W_{\overline{\sigma}}$, where $W_{\overline{\sigma}}$ is already updated when the computation of `ComputeNoWin`($\mathcal{G}$, σ, n, $W_{\overline{\sigma}}$) starts. Similarly to the previous case, we enrich the computation of `ComputeNoWin`($\mathcal{G}$, σ, n, $W_{\overline{\sigma}}$), and remember a failure vertex for each vertex which is not in $W_{\overline{\sigma}}$ and thus will be in W_{tie}. In each iteration of `ComputeNoWin` the vertices added to the computed set are of three types: $X_{\overline{\sigma}}$, $Y_{\overline{\sigma}}$ and Y_{tie}.

$v' \in X_{\overline{\sigma}}$**:** The set $X_{\overline{\sigma}}$ is computed by $\text{Attr?}_{\overline{\sigma}}(\mathcal{G}, nowin)$. Thus in order to find failure vertices for such vertices that are not in $W_{\overline{\sigma}}$ we use an enriched computation of the may-attractor set, as described in the case of $n = 0$. This time in the initialization of the computation we also have the set *nowin* from the previous iteration, for which we already have the required information.

$v' \in Y_{tie}$**:** Vertices in Y_{tie} already have a failure vertex and cause, recorded during the computation of `SolveGame`($\mathcal{G}[Y]$).

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

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

As for the iteration, suppose we have $\mathrm{Attr?}^i_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$, with the additional information attached to every vertex in it which is in $Y_{\overline{\sigma}}$. We now compute the set $\mathrm{Attr?}^{i+1}_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$. Let v' be a vertex that is added to $\mathrm{Attr?}^{i+1}_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$. If $v' \notin Y_{\overline{\sigma}}$, then no information is needed. Otherwise, we do the following.

p1. If $v' \in V_\sigma$ and it has a may edge $v'E^-v''$ which is *not* a must edge to $v'' \in \bar{Y} \setminus W_{\overline{\sigma}}$, then v' is a failure vertex, with this edge being the cause [failEsc].

p2. If $v' \in V_\sigma$ and it has a must edge to $v'' \in X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$, then we set the failure vertex and cause of v' to be those of v'' (which are already computed).

p3. Otherwise, v' has a may (possibly must) edge to a vertex $v'' \in \mathrm{Attr?}^i_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}}) \cap Y_{\overline{\sigma}}$. In this case the failure vertex and cause of v' are those of v''.

The intuition behind the failure identification for $v' \in Y_{\overline{\sigma}}$ is as follows. In case p1, v' is considered a failure vertex (case [failEsc]), with the may (not must) edge to $v'' \in \bar{Y} \setminus W_{\overline{\sigma}}$ being the cause. This is because the existence of this edge allows Player σ to "escape" from $Y_{\overline{\sigma}}$ where Player $\overline{\sigma}$ wins in $\mathcal{G}[Y]$ into $\bar{Y} \setminus W_{\overline{\sigma}}$ where he does not win. If this edge did not exist, v' would not be added to the may-attractor set of σ, and thus would remain in $W_{\overline{\sigma}}$ in $\mathcal{G}$. A careful analysis (see Lemma 4.17 below) shows that the only possibility where there exists such a *must* edge to $v'' \in \bar{Y} \setminus W_{\overline{\sigma}}$ is when this edge is to $X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$ (recall that $X_{\overline{\sigma}} \subseteq \bar{Y}$, as illustrated by Figure 4.1). This possibility is handled separately in case p2. The set $X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$ is a subset of W_{tie} for which the failure was already analyzed, and in case p2 we set the failure vertex and cause of v' to be those of $v'' \in X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$. This is because changing the classification of v'' to $W_{\overline{\sigma}}$ would make a step in the direction of changing the classification of $v' \in V_\sigma$ to $W_{\overline{\sigma}}$ as well. Similarly, since the edge from $v' \in V_\sigma$ to v'' is a must edge, changing the classification of v'' to W_σ would change the classification of v' to W_σ. In all other cases, the computation recursively continues with a vertex in $Y_{\overline{\sigma}}$ that was already added to the may-attractor set and that *affects* the addition of v' to it (case p3).

Lemma 4.17. *The computation of failure vertices for $n \geq 1$ is well defined, meaning that all the possible cases are handled.*

Proof. To be convinced of this, one needs to notice that when $v' \in Y_{\overline{\sigma}}$ all the possible cases are handled. By the definition of $\mathrm{Attr?}_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$, every vertex $v' \in Y_{\overline{\sigma}}$ that is added to the set $\mathrm{Attr?}^{i+1}_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$ fulfills one of the following possibilities.

- $v' \in V_\sigma$ and has a may edge to a vertex $v'' \in \mathrm{Attr?}^i_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$. We have three possibilities.
  - If there exists such an edge which is *not* a must edge and for which $v'' \in \bar{Y} \setminus W_{\overline{\sigma}}$, then case p1 applies.
  - If v' has a *must* edge to a vertex $v'' \in \bar{Y} \setminus W_{\overline{\sigma}}$, then we have the following. We first show that it must be the case that $v'' \in X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$. Note that $\bar{Y} \setminus W_{\overline{\sigma}} = (X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}) \cup (\mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], N) \setminus W_{\overline{\sigma}})$. Thus it suffices to show that

$v'' \notin \mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], N) \setminus W_{\overline{\sigma}}$. Assume the contrary. We know that $v' \in Y_{\overline{\sigma}} \subseteq Y \subseteq X_\sigma$. Therefore we get that $v' \in \mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], \mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)) = \mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], N)$ (Lemma 4.1). Yet, we have that $\mathrm{Attr!}_\sigma(\mathcal{G}[X_\sigma], N) = X_\sigma \setminus Y$, in contradiction to the fact that $v' \in Y$. We conclude that $v'' \in X_{\overline{\sigma}} \setminus W_{\overline{\sigma}}$ and case p2 applies.

  - Otherwise, v' has a may edge to $v'' \in \mathrm{Attr?}^i_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}}) \cap Y$. Since $v' \in V_\sigma$ is in $Y_{\overline{\sigma}}$ all its successors in Y have to be in $Y_{\overline{\sigma}}$ as well. This is true in particular for $v'' \in Y$. Thus, it must be the case that $v'' \in Y_{\overline{\sigma}}$ and case p3 applies.

- $v' \in V_{\overline{\sigma}}$ and all its must edges are to vertices in $\mathrm{Attr?}^i_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$. Since $v' \in Y_{\overline{\sigma}}$, then at least one of these must edges has to be to a vertex $v'' \in Y_{\overline{\sigma}}$ as well. Thus case p3 applies.

□

This concludes the description of how `SolveGame` records the failure information for each vertex in W_{tie}. Altogether, the failure vertex is classified as one of four types: failDE, failP$\overline{\sigma}$, failPσ (identified during the enriched computation of $\mathrm{Attr?}_{\overline{\sigma}}(\mathcal{G}, nowin)$), or failEsc (identified during the enriched computation of $\mathrm{Attr?}_\sigma(\mathcal{G}, \bar{Y} \setminus W_{\overline{\sigma}})$). A simple case analysis shows the following.

Theorem 4.18. *Let v_f be a vertex that is classified by* `SolveGame` *as a failure vertex. The failure cause can either be the fact that $v_f \in V_{tie}$, or it can be a genuine may edge $(v_f, v'') \in E^- \setminus E^+$.*

Example 4.19. We illustrate the failure identification on the 3-valued parity game $\mathcal{G}$ depicted in Figure 3.3, where $v_{00} \in W_{tie}$ (see Example 4.14). We follow the execution of `SolveGame`($\mathcal{G}$) and describe the way it records a failure vertex and a failure cause for each vertex v which is added to $W_{tie} = V \setminus \{v_{08}\}$. We denote the failure cause by $f(v)$. Figure 4.6 visualizes the process of failure identification. Namely, the failure vertices and their failure causes are depicted in boldface. Moreover, the edges along which the failure information is propagated from one vertex to another are labeled by f.

In this example $n \geq 1$. Therefore, failure information is recorded by `ComputeNoWin`($\mathcal{G}$, $\sigma = 0$, $n = 2$, $W_1 = \emptyset$) for every new vertex which is added to *nowin*. We follow the addition of vertices to *nowin* as described in Example 4.14, where in each iteration of the loop, X_1, Y_1 and Y_{tie} are added to *nowin*.

In the first iteration, $Y_{tie} = \emptyset$, therefore only X_1 and Y_1 are added to *nowin*:

X_1 := $\mathrm{Attr?}_1(\mathcal{G}, \emptyset) = \{v_{16}, v_{18}\}$. For these vertices, failure information is recorded during the computation of the may-attractor set as follows (see case $n = 0$): In the initialization of the computation, $v_{18} \in D_{tie}$ is added to $\mathrm{Attr?}_1(\mathcal{G}, \emptyset)$. Thus,

$f(v_{18}) := v_{18}$ [failDE]

In the first iteration, $v_{16} \in V_1$ is added since it has a must edge to $v_{18} \in \mathrm{Attr?}^0_1(\mathcal{G}, \emptyset)$. Note that $v_{18} \notin W_1$. Therefore, by case o3, the failure cause of v_{16} is that of v_{18}:

$f(v_{16}) := f(v_{18}) = v_{18}$

$Y_1 := Y = \{v_{12}, v_{13}, v_{14}, v_{15}, v_{17}, v_{02}, v_{07}\}$. For these vertices, failure information is recorded by an enriched computation of $\mathrm{Attr?}_0(\mathcal{G}, \bar{Y} \setminus W_1) = \mathrm{Attr?}_0(\mathcal{G}, \bar{Y})$ (since $W_1 = \emptyset$): In the initialization of the computation, $v_{18} \in D_{tie}$, $v_{08} \in D_1$ and $\bar{Y}$ are added to $\mathrm{Attr?}^0_0(\mathcal{G}, \bar{Y})$, all of which are not in Y_1, thus no information is recorded for them. In the first iteration, $v_{02} \in V_0$ is added since it has a may edge to $v_{03} \in \mathrm{Attr?}^0_0(\mathcal{G}, \bar{Y})$. Similarly, $v_{07} \in V_0$ is added to the may-attractor set because of its may edge to $v_{04} \in \mathrm{Attr?}^0_0(\mathcal{G}, \bar{Y})$. v_{03} and v_{04} are both in $\bar{Y}$. Thus, these may edges allow Player 0 to "escape" (giving up her eagerness) from $v_{02}, v_{07} \in Y_1$ to $\bar{Y}$ where it is no longer ensured that Player 1 wins. Therefore, by case p1:

$f(v_{02}) := (v_{02}, v_{03})$ [failEsc]
$f(v_{07}) := (v_{07}, v_{04})$ [failEsc]

Still in the first iteration, $v_{15} \in V_0$ is added to the may-attractor set since it has a must edge to $v_{16} \in \mathrm{Attr?}^0_0(\mathcal{G}, \bar{Y})$. Here case p1 does not apply since v_{16} is not in $\bar{Y}$. However, it is in X_1, and already has a failure cause. Therefore, by case p2, the failure cause of v_{15} is that of v_{16}, i.e.,

$f(v_{15}) := f(v_{16}) = v_{18}$

In the second iteration of the may-attractor computation, $v_{14} \in V_0$ is added since it has a must edge to v_{15} which is already in the may-attractor set. Similarly, v_{17} and v_{13} are added in the third iteration due to the must edge to v_{14}, and v_{12} is added in the forth iteration due to the must edge to v_{13}. In all cases, the target vertices (v_{15}, v_{14} and v_{13}) are also in Y_1, and already have failure causes from the preceding iteration of the may-attractor computation. Thus case p3 applies and the failure cause is "inherited" from the target vertex. Namely,

$f(v_{14}) := f(v_{15}) = v_{18}$
$f(v_{17}) := f(v_{14}) = v_{18}$
$f(v_{13}) := f(v_{14}) = v_{18}$
$f(v_{12}) := f(v_{13}) = v_{18}$

The remaining vertices of W_{tie} are added to *nowin* in the second iteration of the loop of `ComputeNoWin`, as part of X_1 (since $Y_1 = Y_{tie} = \emptyset$):

$X_1 := \mathrm{Attr?}_1(\mathcal{G}, nowin) = V \setminus \{v_{08}\}$. The vertices that are not already in *nowin* are added to the may-attractor set in the following order: v_{01}, v_{05}, v_{11} in the first iteration, v_{00}, v_{04}, v_{06} in the second iteration, and finally v_{03} in the third iteration. All of them are added as vertices in V_0 for which all of the outgoing may edges are to vertices that are already in the may-attractor set. The target vertices of these may edges are all *not* in W_1, and hence have a failure cause. The failure cause is "inherited" from them (see case p3). Namely,

$f(v_{01}) := f(v_{02}) = (v_{02}, v_{03})$
$f(v_{05}) := f(v_{07}) = (v_{07}, v_{04})$
$f(v_{11}) := f(v_{12}) = v_{18}$
$f(v_{00}) := f(v_{01}) = (v_{02}, v_{03})$
$f(v_{04}) := f(v_{05}) = (v_{07}, v_{04})$
$f(v_{06}) := f(v_{01}) = (v_{02}, v_{03})$
$f(v_{03}) := f(v_{04}) = (v_{07}, v_{04})$

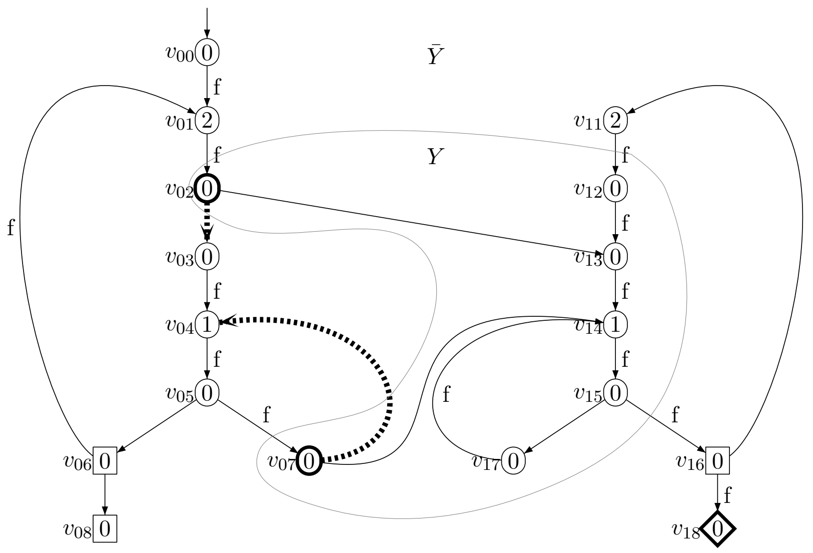


Figure 4.6: Illustration of the failure analysis for the 3-valued parity game $\mathcal{G}$ depicted in Figure 3.3.

In particular, the failure vertex of v_{00} is v_{02} with the failure cause being the may edge (v_{02}, v_{03}). No further vertices are added to *nowin* in the third (and last) iteration.

4.3.2 Failure Analysis

Once we are given a failure vertex $v_f = s'_a \vdash \varphi'$ and a corresponding cause for failure, we guide the refinement to discard the cause for failure in the hope of changing the model checking result to a definite one. This is done as in [74].

For the completeness of the presentation, we repeat here (a simplified version of) the failure analysis of [74]. There, the failure information is used to determine how the set of concrete states represented by s'_a should be split in order to eliminate the failure cause. It is then decided how to alter the abstraction in order to ensure the required split. To solve the latter problem, a criterion for splitting *all* abstract states, while ensuring the split of the failure state s'_a, is found by known techniques, depending on the type of abstraction used. Examples of solutions for certain types of abstractions can be found in [20] (for abstraction based on invisible variables), in [18] (for abstraction based on formulas clusters) and in [64] (for predicate abstraction). It remains to explain which subsets of $\gamma(s'_a)$ need to be separated by the refinement. Let $M_C = (S_C, R, L_C)$ and $M_A = (S_A, R^+, R^-, L_A)$ be the concrete and abstract models at hand. The criterion for the separation depends on the type of the failure vertex v_f and is found by the following analysis.

1. If $v_f \in V_{tie}$, then this means that $v_f = (s'_a, l)$ where $l \in Lit$ is such that both

$l \notin L_A(s'_a)$ and $\neg l \notin L_A(s'_a)$. In this case, the indefinite classification results from the fact that s'_a represents concrete states that are labeled by l as well as concrete states that are labeled by $\neg l$. In order to avoid the indefinite result in this vertex, we need to separate these types of concrete states. Hence, our goal is to separate $\gamma(s'_a)$ to two sets $\{s'_c \in \gamma(s'_a) \mid l \in L_C(s'_c)\}$ and $\{s'_c \in \gamma(s'_a) \mid \neg l \in L_C(s'_c)\}$.

2. If the failure cause is an edge $(v_f, v'') \in E^- \setminus E^+$, where $v'' = (s''_a, \varphi'')$, then the edge is based on a may transition (s'_a, s''_a) of the abstract model which is not a must transition. Let $conc_{must} = \{s'_c \in \gamma(s'_a) \mid \exists s''_c \in \gamma(s''_a) \text{ s.t. } s'_c R s''_c\}$ be the set of all concrete states, represented by s'_a, that have an outgoing transition to a state represented by s''_a. Our goal is to separate the sets $conc_{must}$ and $\gamma(s'_a) \setminus conc_{must}$.

The intuition behind the criterion for the split that is derived from case 1 is clear. Its purpose is to allow us to conclude definite results about the new abstract states obtained by the split of the failure vertex. In case 2, our goal is to obtain a must transition between (parts of) s'_a and s''_a in the abstract model and on the other hand remove the may transition altogether between other parts of s'_a and s''_a. Either way, this makes the abstract model closer to the concrete one.

In fact, in [74], a finer distinction is made in case 2 based on the subformula of v_f and on whether $v'' \in W_0$, $v'' \in W_1$ or $v'' \in W_{tie}$. We omit this discussion here, as well as other optimizations of the failure analysis.

It is possible that one of the sets obtained during the failure analysis is empty and provides no criterion for the split. Yet, new information can be gained from it as well. For example, consider case 2. Suppose $conc_{must} = \gamma(s'_a)$, then in fact every state that is represented by the underlying state s'_a of v_f has an outgoing transition to a state represented by the underlying state s''_a of v'', which means that the transition (s'_a, s''_a) can in fact be added as a must transition to the abstract model, and the corresponding edge (v_f, v'') can be added as a must edge to the game. If $conc_{must} = \emptyset$, then in fact none of the concrete states in $\gamma(s'_a)$ has a transition to a concrete state that is represented by s''_a. Thus, the may transition (s'_a, s''_a) and the corresponding may edge (v_f, v'') can be removed. Similar arguments apply to case 1, where a trivial split criterion will indicate that the labeling function of the abstract model can be updated. Either way, the game can be re-evaluated based on the new information.

Example 4.20. Consider the 3-valued parity game depicted in Figure 3.3 for the model checking problem described in Example 3.2. The failure vertex obtained in this example is v_{02} and its failure cause is the may edge (v_{02}, v_{03}) (see Example 4.19). This may edge results from the may transition (s_0, s_0) of the model (see Figure 2.1). Thus, refinement is performed by case 2 of the failure analysis: In this example, the failure state s_0 represents the concrete states s_{00} and s_{10}. Only s_{00} has an outgoing transition that corresponds to the may transition. Therefore, $conc_{must} = \{s_{00}\}$ and refinement is aimed at separating the sets $\{s_{00}\}$ and $\{s_{10}\}$.

4.4 Incremental Abstraction-Refinement Framework

Having suggested a refinement mechanism that suits the 3-valued model checking algorithm of Section 4.2, we now have all the components required for an iterative abstraction-refinement framework, where each iteration consists of abstraction, model checking and refinement.

Theorem 4.21. *For finite concrete models, iterating the abstraction-refinement process is guaranteed to terminate with a definite answer.*

Proof. Since refinement is done by splitting the abstract states, then applying the refinement on the abstract model results in a refined abstract model with the following properties. Every state s_r in the refined model has some *super-state* s_u in the unrefined model, in the sense that the set of concrete states that s_r represents is a subset of those represented by s_u. Furthermore, since our refinement splits at least one state[2], it ensures that at least one of the refined states represents strictly less concrete states than its super-state. Thus, the number of iterations in the abstraction-refinement process is bounded by the number of concrete states and is guaranteed to end when the state space is finite. □

We refine abstract models by splitting their states. The criterion for the refinement is decided locally, based on one failure vertex, but it has a global effect, since the refinement is applied to the whole abstract model.

After refinement, one has to re-run the model checking algorithm on the refined KMTS to get a definite value for s_c and φ. Yet, in practice, there is no reason to split or re-evaluate states for which the model checking results are definite. As in [74], the game-based model checking algorithm provides us with a convenient framework to use previous results and avoid unnecessary refinement. Namely, we can restrict the re-evaluation process to the previous W_{tie}: When constructing the 3-valued parity game based on the refined KMTS, every vertex $s_a^2 \vdash \varphi'$ for which a vertex $s'_a \vdash \varphi'$ (where s_a^2 results from splitting s'_a) exists in W_0 or W_1 from the previous iteration can be considered a dead-end winning for Player 0 or Player 1, respectively. This way we avoid unnecessary refinement. The result is an *incremental* model checking algorithm based on iterative abstraction-refinement.

4.5 Concluding Remarks

This chapter presents a game-based model checking for abstract models w.r.t. specifications in μ-calculus, interpreted over a 3-valued semantics, together with automatic refinement, if the model checking result is indefinite. Model checking is performed by a new algorithm for solving the 3-valued parity game defined in Chapter 3. The refinement is based on a failure analysis that follows the run of the algorithm.

While the time complexity of our algorithm for solving the 3-valued game is exponential in the alternation depth d of the formula to check, Jurdzinski [47] presented an algorithm for solving ordinary parity games that is exponential in $d/2$. It would be interesting to find out whether this algorithm could be adapted to the 3-valued setting with automatic refinement.

[2]In fact, as explained in Section 4.3.2, a refinement step might not split any state, but in such a case it will update the labeling or the transitions of the abstract model. These updates are also limited to a finite number since after at most a number of them which is bounded by the size of the abstract model an exact abstract model will be obtained.

Chapter 5

Game-Based Abstraction-Refinement: Reduction Approach

5.1 Introduction

Refinement of indefinite model checking results for the μ-calculus w.r.t. abstract models has been suggested in the previous chapter, following the refinement of [74] for CTL. In both cases, the refinement is based on finding a cause for the indefinite result by following the run of an algorithm that solves a corresponding 3-valued model checking game. Being based on an especially tailored algorithm, a similar approach is not applicable when the 3-valued model checking game is solved via a reduction to two 2-valued model checking games.

In this chapter we present a novel approach, which shows that refinement information can be extracted from two 2-valued model checking games, provided that they are defined over the full game board of the 3-valued game. Our refinement is based on the new notion of *non-losing* rather then winning strategies. This approach is beneficial since it can take advantage of any game-based model checking algorithm for the μ-calculus w.r.t. the 2-valued semantics [32, 47].

We will now explain our new approach in more detail. Recall that in the 3-valued model checking game for the μ-calculus defined in Chapter 3, a new possibility of a *tie* is added. Therefore, we can now consider for each player, in addition to a winning strategy also a *non-losing* strategy, which guarantees that each play will end with either a win for this player or a tie, no matter what the other player does.

To simplify the presentation, we consider the equivalent formulation of the game as a 3-valued parity game with players 0 and 1 (see Chapter 3). In order to determine the winner, if there is one, we reduce this game to two ordinary (2-valued) parity games, $\mathcal{G}_0$ and $\mathcal{G}_1$. Player 0 has a winning strategy on game $\mathcal{G}_0$ iff Player 0 has a winning strategy on the original 3-valued game $\mathcal{G}$. Furthermore, Player 0 has a winning strategy on $\mathcal{G}_1$ iff she has a non-losing strategy on $\mathcal{G}$. The dual facts hold for Player 1.

When the game $\mathcal{G}$ results in a tie, and a refinement is needed, non-losing strategies become extremely helpful. In this case none of the players have a winning strategy, which means that considering winning strategies does not provide a witness for the tie result. Non-losing strategies, however, take exactly this role: when the result is a tie, each player has a non-losing strategy (which corresponds to a winning strategy on one of the 2-valued games). These strategies can be combined to one play along which a cause for the tie can be found. A refinement criterion is then suggested and abstract states are refined (split) accordingly. The refinement itself is performed as in Chapter 4.

We note that the 3-valued model checking game still has an important role. Namely, a similar approach for refinement is not applicable when the 3-valued model checking problem itself is reduced to two 2-valued model checking problems (e.g. [39]), each solved by a separate 2-valued game. This is because then each of the 2-valued games considers a different part of the game board: one considers the part required for proving the formula, while the other considers the part required for proving its negation. For refinement purposes, on the other hand, it is important to consider the *full* game board of the 3-valued game (see Section 5.3 for more details).

The result of this chapter is an incremental abstraction-refinement scheme for the μ-calculus, which resembles the scheme developed in Chapter 4, but uses a reduction for the purpose of solving the model checking game, and uses a novel approach for determining a cause for the tie in case the model checking result is indefinite. The rest of this chapter is devoted to the description of these two ingredients.

5.1.1 Related Work

In Chapter 4 we have suggested a game-based abstraction-refinement framework for the μ-calculus, where the model checking algorithm is based on a direct algorithm for solving the 3-valued game. The current chapter presents an alternative abstraction-refinement framework where the model checking algorithm is based on a reduction of the 3-valued game. It can therefore take advantage of any algorithm for solving ordinary parity games, which makes it more general than the algorithm presented in Chapter 4 which generalizes Zielonka's algorithm for solving parity games. The refinement mechanisms of the two approaches are incomparable.

The reduction approach to solving the 3-valued game described in this chapter is reminiscent of the reduction approach to 3-valued model checking used for example in [45, 37, 39]. Still, in this chapter we reduce the 3-valued *game* into two 2-valued games, rather than reducing the model checking *problem*. As explained above, this is a crucial difference when it comes to refinement. In particular, the works on 3-valued model checking via reduction do not suggest an automatic refinement, whereas the work described in this chapter does.

The model checking algorithm used in our recent work [35] on 3-valued game-based abstraction-refinement also solves the 3-valued game via reduction, similarly to the algorithm in this chapter. Still, the algorithms differ in their refinement since in [35], a criterion for refinement is found by heuristics, independently of the model checking algorithm. The relevance of the chosen criterion for the tie result is therefore less clear than the criterion chosen in this chapter which is based on non-losing strategies for the players.

For more details on these works, as well as further related work, we refer the reader to the related work reported in Chapter 4.

5.2 Solving 3-Valued Parity Games via Reduction

In this section, we discuss solving 3-valued parity games via a reduction to ordinary parity games. We compute the winning sets W_0 and W_1 of the players, as well as W_{tie}, where none of the players has a winning strategy (see Chapter 4).

It is not hard to see that solving a 3-valued parity game $\mathcal{G}$ can be reduced to solving two ordinary parity games: first try to find a winning strategy for Player 0 disregarding her genuine may edges (i.e., may edges which are not must edges) and treating dead-end vertices in V_{tie} as losing for her. If the result is negative then try to find a winning strategy for Player 1 disregarding his genuine may edges and treating tie dead-ends as losing for him. This is reminiscent of the approach of [11], where a 3-valued interpretation of a formula in a partial model is computed by considering a pessimistic and an optimistic interpretation.

More precisely, in order to find the winning set of Player σ, we reduce $\mathcal{G}$ into an ordinary parity game denoted $\mathcal{G}_\sigma$ by (1) removing all the outgoing genuine may edges of vertices of Player σ, (2) ignoring the distinction between may and must edges in the remaining edges, and (3) adding V_{tie} to V_σ (meaning that Player $\overline{\sigma}$ wins in these vertices). Note that we do not change the set of vertices, nor the priority function. Formally, $\mathcal{G}_\sigma$ is defined as follows.

Definition 5.1. *Let $\mathcal{G} = (A, \Theta)$ be a 3-valued parity game with arena $A = (V_0, V_1, V_{tie}, E^+, E^-)$. For $\sigma \in \{0, 1\}$, the* σ-reduced game *is an ordinary parity game $\mathcal{G}_\sigma = (A_\sigma, \Theta)$ with arena $A_\sigma = (V_0^\sigma, V_1^\sigma, E)$, where $V_\sigma^\sigma = V_\sigma \cup V_{tie}$, $V_{\overline{\sigma}}^\sigma = V_{\overline{\sigma}}$, and $E = E^- \setminus \{(v, v') \mid v \in V_\sigma$ and $(v, v') \notin E^+\}$.*

$\mathcal{G}_\sigma$ might contain dead-end vertices, some of which result from dead-end vertices in $\mathcal{G}$ and some result from vertices of V_σ that had only genuine may edges in $\mathcal{G}$. This means that they become dead-ends in $\mathcal{G}_\sigma$. However, $\mathcal{G}_\sigma$ can be transformed into a game whose underlying graph is total as described in Remark 3.14.

Proposition 5.2. *Let $\mathcal{G}$ be a 3-valued parity game. Then Player σ has a winning strategy from set $V' \subseteq V$ in $\mathcal{G}$ iff she has a winning strategy from V' in $\mathcal{G}_\sigma$. Moreover, a winning strategy for Player σ from V' in $\mathcal{G}_\sigma$ is also a winning strategy for her in $\mathcal{G}$.*

We conclude that for $\sigma \in \{0, 1\}$, the winning set of Player σ in $\mathcal{G}$, W_σ, is exactly the winning set of Player σ in $\mathcal{G}_\sigma$ and $W_{tie} = V \setminus (W_0 \cup W_1)$. Therefore, solving the 3-valued parity game reduces to solving the two ordinary parity games $\mathcal{G}_0$ and $\mathcal{G}_1$:

```
Algorithm SolveThreeValuedGame ($\mathcal{G}$)
  1. ($W_0^0$, $W_1^0$) := SolveOrdinaryGame ($\mathcal{G}_0$);
  2. ($W_0^1$, $W_1^1$) := SolveOrdinaryGame ($\mathcal{G}_1$);
  3. ($W_0$, $W_1$, $W_{tie}$) := ($W_0^0$, $W_1^1$, $V \setminus (W_0^0 \cup W_1^1)$);
  4. return ($W_0$, $W_1$, $W_{tie}$);
```

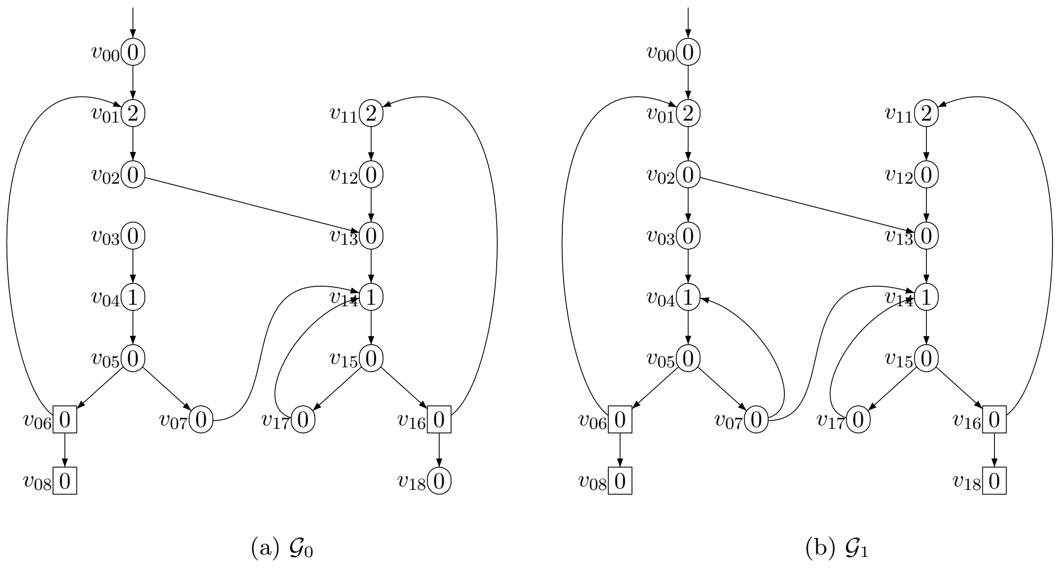


Figure 5.1: The reduced games of the parity game from Figure 3.3.

Solving $\mathcal{G}_0$ and $\mathcal{G}_1$ can be done using any of the existing algorithms for solving ordinary parity games, maintaining their complexity. Moreover, winning strategies in the 3-valued game can be easily obtained from winning strategies in the ordinary games, since a winning strategy for Player σ in $\mathcal{G}_\sigma$ is also a winning strategy for Player σ in $\mathcal{G}$.

Remark 5.3. Note that even if the graph of the original game $\mathcal{G}$ is connected, the underlying graph of $\mathcal{G}_\sigma$ might not be connected. Thus, depending on the algorithm for solving ordinary parity games, if we wish to classify *all* the vertices of $\mathcal{G}$, it might be necessary to invoke the algorithm for every connected component in $\mathcal{G}_\sigma$ separately (for example, if the algorithm has an on-the-fly nature and it considers only reachable vertices).

Example 5.4. The reduction of the 3-valued parity game depicted in Figure 3.3 to two games, $\mathcal{G}_0$ and $\mathcal{G}_1$ is shown in Figure 5.1. Note that genuine may edges of Player 0 vertices are removed in $\mathcal{G}_0$ and that v_{18} is declared as a Player 0 vertex in $\mathcal{G}_0$ (meaning that Player 1 wins in it) and a Player 1 vertex in $\mathcal{G}_1$ (meaning that Player 0 wins in it).

We see that the only vertex from which Player 0 has a winning strategy in $\mathcal{G}_0$ is v_{08}. In $\mathcal{G}_1$, Player 1 has no winning strategy regardless of which vertex the game starts in. We conclude that $W_0 = \{v_{08}\}$, $W_1 = \emptyset$, and that all remaining vertices form the set W_{tie}.

5.3 Refinement

When the 3-valued parity game at hand is in fact a 3-valued model checking game $\Gamma_M(s_a, \varphi)$, solving the game as described in the previous section results in a model checking algorithm. In this section we discuss refinement that suits the above model checking algorithm.

Recall that refinement is needed if the vertex of interest $v = s_a \vdash \varphi$ in the 3-valued parity game is in W_{tie}. This means that a refinement step is required if none of the players has a winning strategy from v. Based on this property, it was stated in [74] as well as Chapter 4 that rather than calling a 2-valued parity games solver twice it is more helpful for refinement purposes to combine the two runs. This is because combining both runs carries more information about the cause for the lack of winning strategies.

But even in a combined fashion of two ordinary runs, the above approach in which the algorithm looks for winning strategies bears a significant disadvantage: if the algorithm looks for *winning strategies* it produces witnesses for its answer only in those cases in which no refinement is needed. Some notion of witness to its answer is, however, needed if the answer is that none of the players has a winning strategy. This is why we suggest to consider *non-losing strategies* instead.

5.3.1 Using Non-Losing Strategies to Solve the Game

Lemma 5.5. *Let $\mathcal{G}$ be a 3-valued parity game. Player σ has a non-losing strategy from v in $\mathcal{G}$ iff player $\overline{\sigma}$ does not have a winning strategy from v in $\mathcal{G}$.*

The first direction of the lemma is quite clear as it is impossible that Player σ has a non-losing strategy and at the same time Player $\overline{\sigma}$ has a winning strategy. For the other direction we use the following proposition that also provides a construction of a non-losing strategy for Player $\overline{\sigma}$ in case Player σ does not have a winning strategy. Recall that in order to compute winning sets and strategies in a 3-valued parity game $\mathcal{G}$ we considered in Section 5.2 the reduced (ordinary) games $\mathcal{G}_0$ and $\mathcal{G}_1$. We now note that the same approach can also be used to compute non-losing strategies for the players. This is formalized by the following proposition, which is in a sense the dual of proposition 5.2.

Proposition 5.6. *Let $\mathcal{G}$ be a 3-valued parity game. Then Player σ has a non-losing strategy from $V' \subseteq V$ in $\mathcal{G}$ iff she has a winning strategy from V' in $\mathcal{G}_{\overline{\sigma}}$. Moreover, a winning strategy for Player σ from V' in $\mathcal{G}_{\overline{\sigma}}$ is in itself a non-losing strategy for her in $\mathcal{G}$.*

We now return to the proof of Lemma 5.5.

Proof of Lemma 5.5. Proposition 5.6 states that Player σ has a non-losing strategy from v in $\mathcal{G}$ iff she has a winning strategy from v in $\mathcal{G}_{\overline{\sigma}}$. By determinacy of ordinary parity games this happens iff Player $\overline{\sigma}$ does not have a winning strategy from v in $\mathcal{G}_{\overline{\sigma}}$ and by Proposition 5.2 this is iff Player $\overline{\sigma}$ does not have a winning strategy from v in $\mathcal{G}$. This concludes the proof of Lemma 5.5. □

Lemma 5.5 implies that a non-losing strategy for a player can be used as a witness to explain why the opponent does not win. Moreover, unlike winning strategies, where it is possible that no player has one, the above lemma implies that at least one player has a non-losing strategy, thus such an explanatory information always exists (at least for one player). This is formalized in the following lemma.

Lemma 5.7. *Let $\mathcal{G}$ be a 3-valued parity game and v a vertex in the game. At least one of the players has a non-losing strategy from v in $\mathcal{G}$.*

Proof. Suppose none of the players has a non-losing strategy from v in $\mathcal{G}$. According to Lemma 5.5 both players would have to have a winning strategy in the game $\mathcal{G}$ which is clearly impossible. □

Lemma 5.7 holds the key for refinement: if we use an algorithm that computes non-losing strategies then we will always have a witness.

Furthermore, Lemmas 5.5 and 5.7 also provide an alternative approach for solving the 3-valued game by considering non-losing strategies rather than winning strategies, as they imply that for a 3-valued parity game:

1. $v \in W_\sigma$ iff only Player σ has a non-losing strategy from v.
2. $v \in W_{tie}$ iff both players have non-losing strategies from v.

In particular, Proposition 5.6 used in the proof of Lemma 5.5 provides a way to compute non-losing strategies using the reduction approach: to compute a strategy that is non-losing for Player σ from $V \setminus W_{\overline{\sigma}}$ in $\mathcal{G}$ we compute a strategy that is winning for Player σ from $V \setminus W_{\overline{\sigma}}$ in the ordinary game $\mathcal{G}_{\overline{\sigma}}$.

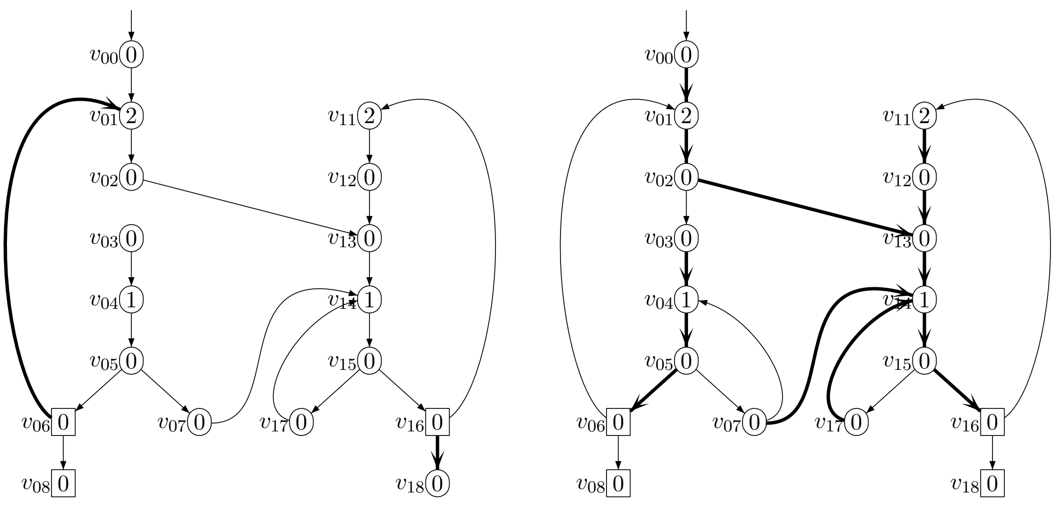
Thus, in comparison to the previous reduction approach (from Section 5.2), we use here the same reduced ordinary parity games, but now in the reduced game $\mathcal{G}_\sigma$, we are interested in a winning strategy of Player $\overline{\sigma}$ rather than of σ. As stated earlier, this approach is particularly helpful when refinement is needed. Here again it might be necessary to invoke the solver of each ordinary parity game several times in case the resulting graph is not connected (see Remark 5.3).

Example 5.8. Let us reconsider the games shown in Figure 5.1, where we are now interested in non-losing (rather than winning) strategies of the players in the original game. In $\mathcal{G}_0$, Player 1 has a winning strategy from all the vertices except v_{08}, which is shown in Figure 5.2(a) by bold edges. This strategy constitutes a non-losing strategy for him in the original game $\mathcal{G}$ from the same vertices. Similarly, Player 0 has a strategy to win every play in $\mathcal{G}_1$, regardless of where the play starts (see Figure 5.2(b)). Consequently, she also has a non-losing strategy from every vertex in the original game $\mathcal{G}$. We conclude that v_{08}, for which only Player 0 has a non-losing strategy in $\mathcal{G}$, is in W_0, whereas the rest of the vertices are in W_{tie} since both players have non-losing strategies from them. As can be expected, this is consistent with Example 3.12, where winning strategies were considered.

5.3.2 Refinement with Non-Losing Strategies

When using an algorithm that solves the 3-valued parity game by computing non-losing strategies, refinement is needed in the case where both players have non-losing strategies from $v = s_a \vdash \varphi$ (meaning that $v \in W_{tie}$).

Our refinement follows the refinement described in Chapter 4. In particular, it is based on a *failure vertex* and a *failure cause*, as defined in Section 4.3. The difference lies in the algorithm for identifying the failure vertex and cause.



(a) $\mathcal{G}_0$ (b) $\mathcal{G}_1$

Figure 5.2: Winning strategies of the reduced games of the parity game from Figure 3.3.

Identifying a Failure Cause

Recall that from each vertex in W_{tie} both players have non-losing strategies. They can be combined into one strategy for each player. Thus, each player $\sigma \in \{0, 1\}$ has a strategy that is non-losing for him from each vertex in $V \setminus W_{\overline{\sigma}}$ and in particular from W_{tie}. Let ζ_0 and ζ_1 be the corresponding strategies of Player 0 and Player 1 respectively. These strategies can be computed using the reduction approach, as explained in Section 5.3.1.

We use the non-losing strategies ζ_0 and ζ_1 for the failure search. Our failure search basically follows the unique play obtained by letting the players play against each other using their non-losing strategies until it identifies a failure vertex v' and a *cause* for the failure. More specifically, the failure search proceeds from one tie-vertex (i.e. vertex in W_{tie}) to the next along this play, guided by the non-losing strategies: from $v \in V_\sigma$ it proceeds to $\zeta_\sigma(v)$. This continues until one of three possibilities occurs:

1. The search reaches a (dead-end) vertex in V_{tie}.

2. The search reaches a vertex in W_σ for $\sigma \in \{0, 1\}$.

3. The search reaches a vertex that was already visited.

Note the following facts regarding the play:

- The play is uniquely determined by ζ_0 and ζ_1.

- The play is a tie, as it is non-losing for both players (conforms to a non-losing strategy of each of them).

- The play is a simple regular path if $|\mathcal{G}| = n < \infty$ is finite, and one of the three possibilities occurs after at most n steps in this play.

Now, in the first possibility $v \in V_{tie}$ is considered a failure vertex, since changing its classification to V_0 or V_1 (by splitting it) would make one of the players closer to winning.

As for the second and third possibilities, in each of them there exists one player that is "closer" to winning the play. In the second possibility this is Player σ, for which the play reached a vertex in W_σ. In the third possibility this is the player σ that corresponds to the parity of the maximal priority that appears in the loop that results from the two occurrences of the same vertex. Note that having identified a loop in the play means that the rest of the play will be an infinite unwinding of the loop. Thus, the maximal priority that will occur infinitely often in the play will be the maximal priority that appears on the loop, whose parity corresponds to σ.

Having that Player σ is "closer" to winning the play, and yet knowing that the play is a tie, implies that there has to exist a genuine may edge used by Player σ in the prefix of the play (otherwise Player σ would win). All of these genuine may edges of player σ are candidates to be considered a failure cause with their source vertex being the failure vertex. This is because changing the may edge into a must edge (by splitting the source vertex) would make Player σ closer to winning and on the other hand removing the edge altogether would make Player $\overline{\sigma}$ closer to winning. The choice of one failure vertex from this set of candidates is a matter of heuristics.

To sum up, given a partition of the vertices of $\mathcal{G}$ to (W_0, W_1, W_{tie}) and given non-losing strategies ζ_0 and ζ_1 for Player 0 and 1 resp., the algorithm `FindFailure`(v) returns a failure vertex v_f and cause for $v \in W_{tie}$.

Algorithm `FindFailure` (v)

```
1. if $v \in V_{tie}$ then return $(v, tie)$;

2. else if $v \in W_\sigma$ then return choose($visited \cdot v$, $\sigma$);

3. else if $v \in visited$ then return choose($visited \cdot v$, parity($visited$, $v$));

4. else      // continue with the search
     add $v$ to $visited$;
     let $\sigma$ be such that $v \in V_\sigma$;
     let $w := \zeta_\sigma(v)$;
     FindFailure($w$);
```

where the function `parity`$(sequence, v)$ returns 0 if the maximal priority that appears in the sequence starting from the vertex v is even, and 1 if it is odd. The function `choose`$(sequence, \sigma)$ chooses a vertex from V_σ that appears in the sequence and has a genuine may edge to its successor in the sequence. It returns the chosen vertex and the corresponding may edge.

This concludes the description of how `FindFailure` looks for a failure vertex and cause. A simple case analysis shows the following.

Theorem 5.9. *Let v_f be a vertex that is returned by* `FindFailure`*(v) as a failure vertex. The failure cause can either be the fact that $v_f \in V_{tie}$, or it can be a genuine may edge $(v_f, v') \in E^- \setminus E^+$.*

Given a failure vertex, we perform the refinement as described in Section 4.3.

Example 5.10. Reconsider the game shown in Figure 3.3, where $v_{00} \in W_{tie}$ (see Examples 3.12 and 5.8), and the non-losing strategies of the players discussed in Example 5.8 (see Figure 5.2). Following both non-losing strategies in $\mathcal{G}$ will guide the play starting in v_{00} to vertex v_{18} through non-winning vertices for both players. Consequently, the state underlying v_{18} should be refined, which is s_1 (see Figure 3.2).

Now, assume that v_{18} is a Player 0 vertex (for example after refinement). This means that v_{18} is now winning for Player 1 in $\mathcal{G}$. As before, Player 1 has a winning strategy in $\mathcal{G}_0$, and thus a non-losing strategy in $\mathcal{G}$, by forcing the play to v_{18}. But also Player 0 can still win in $\mathcal{G}_1$, and thus not lose in $\mathcal{G}$, by choosing in v_{02} the edge leading to v_{03} (instead of the edge leading to v_{13}). Now, the combined non-losing strategies would give a loop $v_{00}v_{01}v_{02}v_{03}v_{04}v_{05}v_{06}v_{01}$, which asks for refining the may edge from v_{02} to v_{03}.

This also demonstrates the importance, in terms of the refinement, of not limiting the computation of winning strategies in the reduced graphs $\mathcal{G}_0$ and $\mathcal{G}_1$ to vertices that are reachable from the vertex of interest. Namely, v_{03} is unreachable in $\mathcal{G}_0$ from v_{00}. Yet, the non-losing strategy of Player 0 takes the play to v_{03}. Thus, the information about a non-losing strategy of Player 1 from v_{03} is essential in order to follow the tie play that guides the refinement. This information is only available provided that v_{03} was considered during the computation of a winning strategy for Player 1 in $\mathcal{G}_0$.

5.4 Concluding Remarks

In this chapter we present a game-based abstraction-refinement scheme w.r.t. specifications in the μ-calculus, interpreted over a 3-valued semantics, similarly to Chapter 4.

However, in contrast to Chapter 4, model checking is determined by solving two ordinary (2-valued) parity games. Still, these games are based on the full board of the 3-valued game. This is particularly important for refinement, for which the board of the 3-valued game holds more information than the two boards of the 2-valued games.

The refinement is based on the novel notion of a *non-losing* strategy. In case the model checking result is indefinite, both players have non-losing strategies. Combining these strategies of the two players comprises a play, resulting with a tie. From this play, a failure vertex and a cause are derived and exploited for refinement.

A non-losing strategy for Player σ can easily be extracted by computing a winning strategy for Player σ on the 2-valued game $\mathcal{G}_{\overline{\sigma}}$. This can be done using any algorithm for solving 2-valued model checking games. Thus, our approach can take advantage of efficient algorithms for this problem, such as Jurdzinski's algorithm for parity games [47].

Recently, there has been an active research on completeness and precision of abstractions for branching time logics (e.g. [63, 70, 27, 29, 28]). Various abstract models which are more expressive than KMTSs were suggested. These models add some kind of disjunctiveness to the model: for example, [27] introduces focus operations, and [70, 29] use hyper-transitions (first introduced by [55]) to model the abstract transitions. Some of these models (e.g. [27, 28]) also consider fairness conditions. While fairness requires different techniques (e.g. in order to determine how to refine the fairness conditions), disjunctiveness can be handled by the approach suggested in this research. This simply requires to define a 3-valued model checking game for such models and to encode the game as a 3-valued parity game. An example of the way this is done in the presence of must hyper-transitions appears in Chapter 6.

Chapter 6

Monotonic Abstraction-Refinement

6.1 Introduction

The goal of this chapter is to improve the effectiveness of the 3-valued abstraction-refinement framework for the μ-calculus. We generalize the definition of abstract models (KMTSs) in order to provide a *monotonic* abstraction-refinement framework. The new definition results in more precise abstract models in which more μ-calculus formulas can be proved or disproved. Finally, we adjust the 3-valued game-based abstraction-refinement approach described in the previous chapters to the new monotonic framework.

In order to motivate the main result of this chapter we first recall the definition of KMTSs, used as abstract models in the context of the μ-calculus. Typically, each state of an abstract model represents a set of states of the concrete model. In order to be conservative for the μ-calculus the abstract model contains both *may* transitions (R^-) which over-approximate transitions of the concrete model, and *must* transitions (R^+), which under-approximate the concrete transitions. More specifically, in an (exact) abstract KMTS, for every abstract states s_a and s'_a, $s_a R^- s'_a$ iff there *exists* a concrete state s_c represented by s_a and there *exists* a concrete state s'_c represented by s'_a such that $s_c R s'_c$ ($\exists\exists$-condition). $s_a R^+ s'_a$ iff for *all* s_c represented by s_a there *exists* s'_c represented by s'_a such that $s_c R s'_c$ ($\forall\exists$-condition). On the other hand, abstractions designed for the verification of universal temporal logics, such as ACTL and LTL, require only over-approximated (may) transitions.

Refinements "split" abstract states so that the new, refined states represent smaller subsets of concrete states. In most abstraction-refinement frameworks designed for universal temporal logics such as ACTL and LTL (e.g. [52, 18, 8, 20, 16]), the refined model obtained from splitting abstract states has less (may) transitions. It is therefore *more precise*, in the sense that it satisfies more properties of the concrete model. We call such a refinement *monotonic*.

For the full μ-calculus with the 3-valued semantics, an abstraction-refinement framework has been suggested in the previous chapters. For such a framework, one would expect that after splitting, the number of must transitions will increase as the number of may transitions decreases. Unfortunately, this is not the case. Once a state s'_a is split, the $\forall\exists$-condition that allowed $s_a R^+ s'_a$ might not hold anymore. As a result, the refinement is not monotonic since μ-calculus formulas that had a definite value in the unrefined model may become indefinite.

In [37] this problem has been addressed. They suggest to keep copies of the unrefined states in the refined model together with the refined ones. This avoids the loss of must transitions and guarantees monotonicity. Yet, this solution is not sufficient because the *old* information is still expressed w.r.t. the "unrefined" states and the *new* information (achieved by the refinement) is expressed w.r.t. the refined states. As a result the additional precision that the refinement provides cannot be combined with the old information. This is discussed extensively in Section 6.2.1.

In this work we suggest a different monotonic abstraction-refinement framework which overcomes this problem. For a given set of abstract states, our approach results in a more precise abstract model in which more μ-calculus formulas have a definite value. Moreover, our approach avoids the need to hold copies of the unrefined states.

Inspired by [63], we define a *Generalized Kripke Modal Transition System* (GTS) in which must transitions are replaced by *must hyper-transitions* [55], which connect a single state s_a to a set of states A. A GTS includes $s_a R^+ A$ iff for *all* s_c represented by s_a there *exists* s'_c represented by some $s'_a \in A$ such that $s_c R s'_c$. This weakens the $\forall\exists$-condition by allowing the resulting states s'_c to be "scattered" in several abstract states.

In general, the number of must hyper-transitions might be exponential in the number of states in the abstract model. In practice, optimizations can be applied in order to reduce their number. We suggest an automatic construction of an initial GTS and its successive refined models in a way that in many cases avoids the exponential blowup.

In order to complete our framework, we also adjust for GTSs the 3-valued game-based model checking and the refinement suggested in the previous chapters for KMTSs. Thus, we obtain a monotonic abstraction-refinement framework that is suitable for both verification and falsification of the full μ-calculus.

6.1.1 Related Work

Our new notion of abstract models takes its inspiration from [63], where it is suggested to weaken the condition required from must transitions in order to achieve *completeness* of the abstraction framework. Completeness means that whenever the concrete model satisfies the formula there exists a *finite* abstract model that allows to verify it. Their work refers to the must transitions in an *abstract alternating transition system*, formed by the product of a program with an alternating tree automaton describing the checked property. Our work, on the other hand, addresses the abstract model directly. Thus, it is more general. We also use a different description for the new (weaker) requirement of must transitions. Furthermore, we suggest an abstraction-refinement framework based on the new notion, whereas the discussion in [63] is merely theoretical: It addresses the *completeness* of the abstraction framework and does not suggest any model checking algorithm, nor a refinement mechanism.

This is also the case in [27], where *focus* operations, which resemble our must hyper-transitions are used. Similarly to [63], their primary concern is also completeness of the abstraction framework. Moreover, the focus operations of [27] are used in the evaluation of $\vee$ formulas, whereas our must hyper-transitions are used in the evaluation of the modalities.

In parallel to our work, must hyper-transitions were also used in [29] for the purpose of abstracting two-player turn-based games. There, the logic of interest is the *alternating* μ-calculus. They also use must hyper-transitions to ensure completeness. Yet, unlike [63, 27], they also consider model checking in the presence of hyper-transitions: Their result for *abstract game structures* implies that the model checking problem for GTSs is reducible to concrete model checking in linear time (and logarithmic space) in the size of the GTS. Yet, the GTS itself might be of size exponential in the size of the set of abstract states (due to the existence of hyper-transitions). Thus the overall complexity is exponential. In addition, they do not suggest an automatic refinement.

In [70] we have suggested a similar abstraction-refinement for GTSs, except that we considered the logic CTL and instead of basing the abstraction-refinement on the game-theoretic approach, we used a symbolic 3-valued model checking algorithm and a suitable refinement.

In our more recent work [35], we have suggested another game-based abstraction-refinement approach where the underlying abstract models are GTSs. The refinement there is more local, and splits only one abstract state. For the split state, *all* must hyper-transitions are calculated, unlike the approach taken in this chapter, where must hyper-transitions are "learned" from the previous iteration. For more details on the differences between the two approaches see Chapter 4.

6.2 Generalized Abstract Models

In this section we suggest the notion of a generalized KMTS and its use as an abstract model which preserves the μ-calculus. This notion allows better precision of the abstraction.

6.2.1 Motivation

The main flaw of using KMTSs as abstract models is in the must transitions, which make the refinement not necessarily *monotonic* w.r.t. the precision preorder.

Definition 6.1 (Precision Preorder)**.** *Let M_1, M_2 be two KMTSs over states S_1, S_2 and let $s_1 \in S_1$ and $s_2 \in S_2$. We say that (M_1, s_1) is* more precise *than (M_2, s_2), denoted $(M_1, s_1) \leq_\mu (M_2, s_2)$, if for every φ in $\mathcal{L}_\mu$: $[\![\varphi]\!]_3^{M_2}(s_2) \neq \perp \Rightarrow [\![\varphi]\!]_3^{M_1}(s_1) = [\![\varphi]\!]_3^{M_2}(s_2)$.*

Similarly, we say that M_1 is more precise *than M_2, denoted $M_1 \leq_\mu M_2$, if for every $\varphi \in \mathcal{L}_\mu$: $M_2 \models \varphi \;\Rightarrow\; M_1 \models \varphi$, and $M_2 \not\models \varphi \;\Rightarrow\; M_1 \not\models \varphi$.*

The following example demonstrates the non-monotonicity problem. We consider the traditional refinement that is based on splitting the states of the (abstract) model.

Example 6.2. Consider the following program P.

```
P::   input: $x > 0$
      pc=1: if $x > 5$ then $x := x + 1$ else $x := x + 2$ fi
      pc=2: while *true* do if $odd(x)$ then $x := -1$ else $x := x + 1$ fi od
```

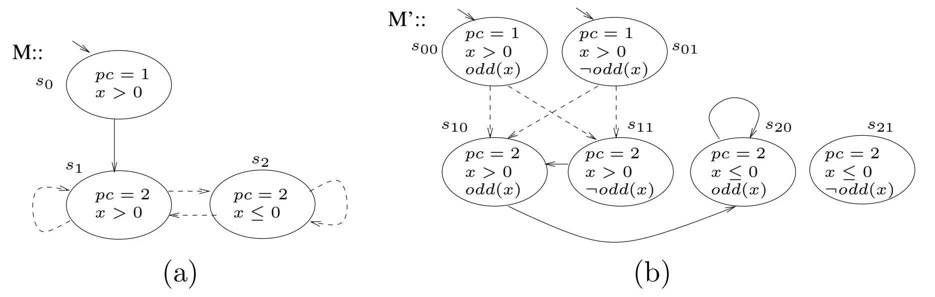


Figure 6.1: (a) An abstract model M describing the program P, and (b) the abstract model M' resulting from its refinement. Outgoing transitions of s_{21} are omitted since they are irrelevant.

Suppose we are interested in checking the property $\varphi = \mu Z.((x \leq 0) \vee \Diamond Z)$ which means "there exists a path in which $x \leq 0$ is eventually satisfied". This property is clearly satisfied by this program. The concrete model of the program is an infinite state model. Suppose we start with an abstract model where concrete states that "agree" on the predicate $(x \leq 0)$ (taken form the checked property φ) are collapsed into a single abstract state. Then we get the abstract model M described in Figure 6.1(a), where the truth value of φ is indefinite. Now, suppose we refine the model by adding the predicate *odd*(x). Then we get the model M' described in Figure 6.1(b), where we still cannot verify φ. Moreover, we "lose" the must transition $s_0 R^+ s_1$ of M. This transition has no corresponding must transition in the refined model M'. This loss causes the formula $\Diamond(x > 0)$ which is true in M to become indefinite in M'. Thus $M' \not\preceq_\mu M$.

The source of the problem is that when dealing with KMTSs as abstract models, we are *not* guaranteed to have a mixed simulation between the refined abstract model and the unrefined one, even if both are exact. This means that the refined abstract model is not necessarily more precise than the unrefined one, even though each of its states represents less concrete states. This is again demonstrated by Example 6.2. There, both the initial states of M' cannot be matched with the (only) initial state s_0 of M in a way that fulfills the requirements of mixed simulation. This is because s_0 has an outgoing must transition whereas the initial states of M' have none. Consequently, $M' \not\preceq M$.

[37] suggests a refinement where the refined model *is* smaller by the mixed simulation than the unrefined one. The solution there is basically to use both the new refined abstract states and the old (unrefined) abstract states. This is a way of overcoming the problem that the target states of must transitions are being split, causing an undesired removal of such transitions. This indeed prevents the loss of precision. Yet, this solution is not sufficient, as demonstrated by the following example.

Example 6.3. Figure 6.2 presents the refined model M'' achieved by applying refinement as suggested in [37] on the model M from Figure 6.1(a). Indeed, we now have a mixed simulation relation from the refined model M'' to the unrefined model M, by simply matching each state with itself or with its super-state, and the loss of precision is prevented. In particular, the truth value of $\Diamond(x > 0)$ in M'' (unlike M' from Figure 6.1(b)) is tt, since there are must transitions from the initial states of M'' to the *old*

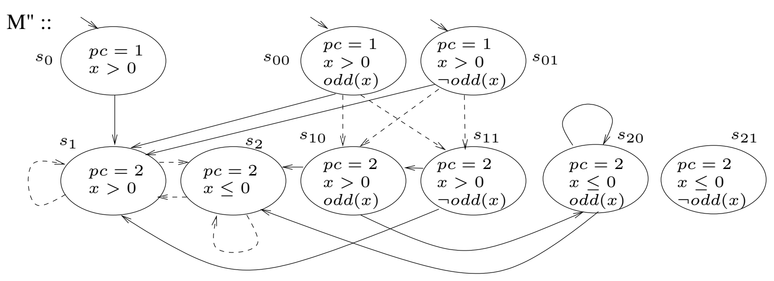

M" ::
s_0
$pc = 1$ $x > 0$
s_{00}
$pc = 1$ $x > 0$ $odd(x)$
$pc = 1$ $x > 0$ $\neg odd(x)$
s_{01}
s_1
$pc = 2$ $x > 0$
s_2
$pc = 2$ $x \leq 0$
s_{10}
$pc = 2$ $x > 0$ $odd(x)$
s_{11}
$pc = 2$ $x > 0$ $\neg odd(x)$
s_{20}
$pc = 2$ $x \leq 0$ $odd(x)$
s_{21}
$pc = 2$ $x \leq 0$ $\neg odd(x)$

Figure 6.2: The model M'' achieved by applying refinement as suggested in [37] on M from Figure 6.1(a). Outgoing transitions of s_{21} are omitted since they are irrelevant, and so are additional outgoing may transitions of the unrefined states (there are no additional outgoing must transitions for the unrefined states).

unrefined state s_1. Yet, in order to verify the desired property $\varphi = \mu Z.((x \leq 0) \vee \Diamond Z)$, we need a must transition to (at least one of) the *new* refined states s_{10} and s_{11} from which a state satisfying $x \leq 0$ is definitely reachable (this information was added by the refinement). However, the $\forall\exists$ condition is still *not* fulfilled between these states. As a result we cannot benefit from the additional precision that the refinement provides and φ is *still* indefinite.

This example demonstrates that even when using the refinement suggested in [37], must transitions may still be absent from the "refined" part of the model, containing the new refined states. As a result the additional precision that the refinement provides cannot necessarily be combined with the old information.

6.2.2 Generalized KMTSs

Having understood the problems that result from the use of must transitions in their current form, our goal here is to suggest an alternative that will allow to weaken the $\forall\exists$ condition. Following the idea presented in [63] (in a slightly different context), we suggest the use of *hyper-transitions* to describe must transitions.

Definition 6.4 (Hyper-Transition)**.** *Given a set of states S, a* hyper-transition *is a pair (s, A) where $s \in S$ and $A \subseteq S$ is a nonempty set of states.*

A (regular) transition (s, s') can be viewed as a hyper-transition (s, A) where $A = \{s'\}$.

Recall that a (regular) must transition exists from s_a to s'_a in an abstract model only if *every* state represented by s_a has a (concrete) transition to *some* state represented by s'_a. The purpose of the generalization is to allow such a concrete transition to exist to *some* state represented by *some* (abstract) state in a set A_a (which plays the role of s'_a). That is, instead of having one abstract state s'_a as the target state and requiring that all the concrete transitions reach concrete states that are represented by s'_a, we will allow the target state to be "scattered" among several states. This can be achieved by using a hyper-transition. The hyper-transition will still perform as a must transition in the sense that it will represent at least one concrete transition of each concrete state represented by s_a (maintaining the $\forall\exists$ meaning).

Definition 6.5. *A* Generalized Kripke Modal Transition System *(***GTS***) is a tuple $M = (S, S^0, R^+, R^-, L)$, where S, S^0, R^- and L are defined as in a KMTS, except that $R^+ \subseteq S \times 2^S$ such that for every $(s, A) \in R^+$, we have that $A \neq \emptyset$ and for each $s' \in A$, $(s, s') \in R^-$ holds.*

The latter requirement replaces the requirement that $R^+ \subseteq R^-$ in a KMTS. A KMTS can be viewed as a GTS where every must hyper-transition consists of a singleton target set.

3-Valued Semantics for GTSs We generalize the 3-valued semantics of $\mathcal{L}_\mu$ for GTSs. The semantics is defined similarly to the 3-valued semantics for KMTSs, except that the use of must transitions is replaced by must hyper-transitions. This affects the definition for formulas of the form $\Box\psi$ or $\Diamond\psi$. Namely,

$$[\![\Box\psi]\!]_3^{M,\rho} \quad := \quad \lambda s. \begin{cases} \text{tt}, & \text{if } \forall t \in S, \text{ if } sR^-t \text{ then } [\![\psi]\!]_3^{M,\rho}(t) = \text{tt} \\ \text{ff}, & \text{if } \exists A \subseteq S \text{ s.t. } sR^+A \text{ and } \forall t \in A : [\![\psi]\!]_3^{M}(t) = \text{ff} \\ \perp, & \text{otherwise} \end{cases}$$

$$[\![\Diamond\psi]\!]_3^{M,\rho} \quad := \quad \lambda s. \begin{cases} \text{tt}, & \text{if } \exists A \subseteq S \text{ s.t. } sR^+A \text{ and } \forall t \in A : [\![\psi]\!]_3^{M}(t) = \text{tt} \\ \text{ff}, & \text{if } \forall t \in S \text{ if } sR^-t \text{ then } [\![\psi]\!]_3^{M,\rho}(t) = \text{ff} \\ \perp, & \text{otherwise} \end{cases}$$

The notion of a mixed simulation relation, that guarantees preservation of μ-calculus formulas between two KMTSs, is generalized as well when dealing with GTSs.

Definition 6.6 (Generalized Mixed Simulation)**.** *Let $M_1 = (S_1, S_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (S_2, S_2^0, R_2^+, R_2^-, L_2)$, be two GTSs, both defined over AP. We say that $H \subseteq S_1 \times S_2$ is a* generalized mixed simulation *from M_1 to M_2 if $(s_1, s_2) \in H$ implies the following:*

1. *$L_2(s_2) \subseteq L_1(s_1)$.*
2. *if $s_1 R_1^- s_1'$, then there is some $s_2' \in S_2$ s.t. $s_2 R_2^- s_2'$ and $(s_1', s_2') \in H$.*
3. *if $s_2 R_2^+ A_2$, then there is some $A_1 \subseteq S_1$ s.t. $s_1 R_1^+ A_1$ and $(A_1, A_2) \in H^{\forall\exists}$, where $(A_1, A_2) \in H^{\forall\exists} \;\Leftrightarrow\; \forall s_1' \in A_1\ \exists s_2' \in A_2 : (s_1', s_2') \in H$.*

If there is a generalized mixed simulation H s.t. $(s_1, s_2) \in H$, then $(M_1, s_1) \preceq (M_2, s_2)$.

If there is a generalized mixed simulation H s.t. $\forall s_1^0 \in S_1^0\ \exists s_2^0 \in S_2^0$ s.t. $(s_1^0, s_2^0) \in H$, and $\forall s_2^0 \in S_2^0\ \exists s_1^0 \in S_1^0$ s.t. $(s_1^0, s_2^0) \in H$, then M_2 is greater by the generalized mixed simulation relation *than M_1, denoted $M_1 \preceq M_2$.*

Thus, the requirements of a generalized mixed simulation are the same as those of a mixed simulation (see Definition 2.4), except that requirement 3 is replaced.

In particular, Definition 6.6 can be applied to a (concrete) Kripke structure M_C and an (abstract) GTS M_A, by viewing the Kripke structure as a GTS where $R^- = R$ and $R^+ = \{(s, \{s'\}) \mid (s, s') \in R\}$. For $(s_c, s_a) \in H$ requirement 3 can be simplified as follows:

3. if $s_a R^+_A A_a$, then there is some s'_c s.t. $s_c R s'_c$ and there is $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$.

This is because requirement 3 requires that for $s_a R^+ A_a$ there exists $s_c R^+ A_c$ such that $(A_c, A_a) \in H^{\forall\exists}$. Yet, when viewing a Kripke structure M_C as a GTS, every must hyper-transition $s_c R^+ A_c$ models a regular transition $s_c R s'_c$, where $A_c = \{s'_c\}$. For such a transition, $(A_c, A_a) \in H^{\forall\exists}$ translates into: for (the single) $s'_c \in A_c$, there exists $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$.

For a Kripke structure the 3-valued semantics agrees with the concrete semantics. Thus, preservation of $\mathcal{L}_\mu$ formulas is guaranteed by the following theorem.

Theorem 6.7. *Let $H \subseteq S_1 \times S_2$ be a generalized mixed simulation relation from a GTS M_1 to a GTS M_2. Then for every $(s_1, s_2) \in H$ we have that $(M_1, s_1) \leq_\mu (M_2, s_2)$. We conclude that if $M_1 \preceq M_2$ then $M_1 \leq_\mu M_2$.*

Proof. The proof is obtained by induction on the structure of μ-calculus formulas, similarly to the proof of Theorem 2.5. The only changes occur in cases where must hyper-transitions are used instead of (ordinary) must transitions:

- Suppose $[\![\Box\psi]\!]_3^{M_2,\rho}(s_2) = \mathrm{ff}$. Then by the definition of the semantics there exists a must hyper-transition from s_2 to A_2 such that for each $s'_2 \in A_2$: $[\![\psi]\!]_3^{M_2,\rho}(s'_2) = \mathrm{ff}$. Moreover, since $(s_1, s_2) \in H$, we know that there exists A_1 such that s_1 has a must hyper-transition to A_1 and $(A_1, A_2) \in H^{\forall\exists}$, meaning that $\forall s'_1 \in A_1\ \exists s'_2 \in A_2 : (s'_1, s'_2) \in H$. Let s'_1 be such a state in A_1 and s'_2 the corresponding state from A_2. Since $s'_2 \in A_2$, we know that $[\![\psi]\!]_3^{M_2,\rho}(s'_2) = \mathrm{ff}$. By the induction hypothesis, this implies that $[\![\psi]\!]_3^{M_1,\rho}(s'_1) = \mathrm{ff}$. That is, $\forall s'_1 \in A_1$: $[\![\psi]\!]_3^{M_1,\rho}(s'_1) = \mathrm{ff}$. Thus $[\![\Box\psi]\!]_3^{M_1,\rho}(s_1) = \mathrm{ff}$.
- The treatment of the case where $[\![\Diamond\psi]\!]_3^{M_2,\rho}(s_2) = \mathrm{tt}$ is dual.

The conclusion that $M_1 \leq_\mu M_2$ is due to the requirement regarding the initial states (as in the case of a mixed simulation relation between KMTSs). □

Construction of an Abstract GTS Given a concrete Kripke structure $M_C = (S_C, S^0_C, R, L_C)$, and an abstraction (S_A, γ) for S_C, an abstract GTS M_A is constructed similarly to an abstract KMTS with the following difference: The must hyper-transitions are computed by a $[\forall\exists\exists]$ rule:

$$\forall s_c \in \gamma(s_a)\ \exists s'_a \in A_a\ \exists s'_c \in \gamma(s'_a) \text{ s.t. } s_c R s'_c \Longleftarrow s_a R^+ A_a$$

This construction assures us that whenever $s_c \in \gamma(s_a)$, then $(M_C, s_c) \preceq (M_A, s_a)$. The generalized mixed simulation $H \subseteq S_C \times S_A$ is induced by γ as follows: $(s_c, s_a) \in H$ iff $s_c \in \gamma(s_a)$ (see the following theorem). Therefore, Theorem 6.7 guarantees preservation of $\mathcal{L}_\mu$ from M_A to M_C.

Theorem 6.8. *Let M_C be a concrete Kripke structure over S_C, and let M_A be a GTS computed as described above based on an abstraction (S_A, γ) for S_C. Then whenever $s_c \in \gamma(s_a)$ then $(M_C, s_c) \preceq (M_A, s_a)$. We conclude that $M_C \preceq M_A$.*

Proof. We show that $H \subseteq S_C \times S_A$ defined by $(s_c, s_a) \in H$ iff $s_c \in \gamma(s_a)$ is a generalized mixed simulation. Let $s_c \in \gamma(s_a)$. Requirements 1 and 2 regarding the labeling and the may transitions are fulfilled as in an abstract KMTS. We now refer to requirement 3. Let $A_a \subseteq S_A$ be such that $s_a R^+ A_a$. By the remark following Definition 6.6, we need to show that there is some $s'_c \in S_C$ s.t. $s_c R s'_c$ and there is $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$. Since $s_a R^+ A_a$, this means (by the construction) that $\forall s_c \in \gamma(s_a)\ \exists s'_a \in A_a\ \exists s'_c \in \gamma(s'_a)$ s.t. $s_c R s'_c$. In particular, for our s_c, we have that $\exists s'_a \in A_a\ \exists s'_c \in \gamma(s'_a)$ s.t. $s_c R s'_c$. By the definition of H, $s'_c \in \gamma(s'_a)$ implies that $(s'_c, s'_a) \in H$. Thus, by changing the order of the existential quantifiers, the requirement holds. The conclusion that $M_C \preceq M_A$ is due to the definition of the initial states (as in a KMTS). $\square$

Other constructions of abstract GTSs can also be suggested. For example, the construction of a mixed transition system from [26] within the framework of abstract interpretation can be extended to GTSs as well.

The use of GTSs allows construction of abstract models that are more precise than abstract models described as KMTSs, when using the same abstract state space and the same concretization function. This is demonstrated by the following example.

Example 6.9. Consider the *exact* KMTS M described in Figure 6.1(a) for the program P from Example 6.2. The state s_1 has no outgoing must transition. Therefore, even verification of the simple formula $\Diamond\Diamond(true)$ fails, although this formula holds in every concrete model where the transition relation is total. Using a GTS (rather than a KMTS) as an abstract model allows us to have a must hyper-transition from s_1 to the set $\{s_1, s_2\}$. Therefore we are now able to verify the formula $\Diamond\Diamond(true)$.

Exact GTS As with KMTSs, the must hyper-transitions of a GTS do not have to be *exact*, as long as they maintain the new $\forall\exists\exists$ condition. That is, it is possible to have less must hyper-transitions than allowed by the $\forall\exists\exists$ rule. If all the components of the GTS are exact, then we get the *exact* GTS, which is *most precise* compared to all the GTSs that are constructed by the same rules based on the given abstraction (S_A, γ).

Optimization Any abstract GTS and in particular the exact GTS can be reduced without damaging its precision, based on the following observation. Given two must hyper-transitions $s_a R^+ A_a$ and $s_a R^+ A'_a$, where $A_a \subset A'_a$, the transition $s_a R^+ A'_a$ can be discarded without sacrificing the precision of the GTS. Therefore, a possible optimization would be to use only *minimal* must hyper-transitions where A_a is minimal. This is similar to the approach of [26], where the target state of a (regular) must transition is chosen to be the smallest state w.r.t. a given partial order on S_A.

In general, even when applying the suggested optimization, the number of must hyper-transitions in the exact GTS might be exponential in the number of states. In practice, computing *all* of them is computationally expensive and unreasonable. Later on, we will suggest how to choose an initial set of must hyper-transitions and increase it gradually in a way that in many cases avoids the exponential blowup.

6.3 Monotonic Abstraction-Refinement Framework

In this section our goal is to show how GTSs can be used in practice within an abstraction-refinement framework designed for full μ-calculus. We also show that using the suggested framework allows us to achieve the important advantage of a *monotonic* refinement when dealing with the full μ-calculus and not just a universal fragment of it.

We start by pointing out that using *exact* GTSs as abstract models solves the problem of the non-monotonic refinement, described in Section 6.2.1.

Definition 6.10 (Split). *Let S_C be a set of concrete states, let S_A and S'_A be two sets of abstract states and let $\gamma : S_A \to 2^{S_C}$, $\gamma' : S'_A \to 2^{S_C}$ be the corresponding concretization functions. We say that (S'_A, γ') is a* split *of (S_A, γ) iff there exists a (total) function $\beta : S'_A \to S_A$ s.t. for every $s_a \in S_A$: $\left(\bigcup_{\beta(s'_a)=s_a} \gamma'(s'_a)\right) = \gamma(s_a)$. If $\beta(s'_a) = s_a$ then s'_a is a* substate *of s_a, and s_a is the* superstate *of s'_a.*

Theorem 6.11 (Monotonicity of GTSs). *Let M_C be a (concrete) Kripke structure and let M_A, M'_A be two* exact *GTSs defined based on abstractions (S_A, γ), (S'_A, γ') respectively, s.t. $M_C \preceq M_A$ and $M_C \preceq M'_A$. If (S'_A, γ') is a split of (S_A, γ), then whenever $s'_a \in S'_A$ is a substate of $s_a \in S_A$ then $(M'_A, s'_a) \preceq (M_A, s_a)$. We conclude that $M'_A \preceq M_A$.*

Proof. Suppose that $s'_a \in S'_A$ is a substate of $s_a \in S_A$. We show that $(M'_A, s'_a) \preceq (M_A, s_a)$. For this purpose we show that $H \subseteq S'_A \times S_A$ defined by $(s'_a, s_a) \in H$ iff $\beta(s'_a) = s_a$ (i.e. s'_a is a substate of s_a) is a generalized mixed simulation. Let $(s'_a, s_a) \in H$. We show that the three requirements hold.

1. Suppose $l \in L_A(s_a)$. Then by the construction scheme, $\forall s_c \in \gamma(s_a) : l \in L_C(s_c)$. Since s'_a is a substate of s_a, then $\gamma'(s'_a) \subseteq \gamma(s_a)$, thus in particular $\forall s_c \in \gamma'(s'_a) : l \in L_C(s_c)$ and by the construction scheme $l \in L_{A'}(s'_a)$. Thus $L_A(s_a) \subseteq L_{A'}(s'_a)$.

2. Suppose $s'_a R^-_{A'} \tilde{s}'_a$. Then by the construction, $\exists s_c \in \gamma'(s'_a)\ \exists s'_c \in \gamma'(\tilde{s}'_a)$ s.t. $s_c R s'_c$. Since s'_a is a substate of s_a, then $\gamma'(s'_a) \subseteq \gamma(s_a)$. Let $\tilde{s}_a \in S_A$ be the superstate of $\tilde{s}'_a$, i.e., $\gamma'(\tilde{s}'_a) \subseteq \gamma(\tilde{s}_a)$. Then in particular $\exists s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(\tilde{s}_a)$ s.t. $s_c R s'_c$. This implies that $s_a R^-_A \tilde{s}_a$. Moreover, $(\tilde{s}'_a, \tilde{s}_a) \in H$ by the definition of H.

3. Suppose $s_a R^+_A A_a$. Then by the construction, $\forall s_c \in \gamma(s_a)\ \exists s_{a1} \in A_a\ \exists s'_c \in \gamma(s_{a1})$ s.t. $s_c R s'_c$, i.e. $\forall s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(A_a)$ s.t. $s_c R s'_c$. Since s'_a is a substate of s_a, then $\gamma'(s'_a) \subseteq \gamma(s_a)$, thus in particular $\forall s_c \in \gamma'(s'_a)\ \exists s'_c \in \gamma(A_a)$ s.t. $s_c R s'_c$. Let $A'_a \subseteq S'_A$ be the set consisting of all the substates of states in A_a. By definition of a split, $\left(\bigcup_{\beta(s'_a)=s_a} \gamma'(s'_a)\right) = \gamma(s_a)$, meaning that $\gamma'(A'_a) = \gamma(A_a)$. Thus the following holds: $\forall s_c \in \gamma'(s'_a)\ \exists s'_c \in \gamma'(A'_a)$ s.t. $s_c R s'_c$. This implies that $s'_a R^+_{A'} A'_a$. Moreover, $(A'_a, A_a) \in H^{\forall\exists}$ since for every $s'_{a1} \in A'_a$, at least one of its superstates s_{a1} is in A_a (otherwise s'_{a1} would not be included in A'_a), and as such $(s'_{a1}, s_{a1}) \in H$. Thus $\forall s'_{a1} \in A'_a\ \exists s_{a1} \in A_a : (s'_{a1}, s_{a1}) \in H$.

To conclude that $M'_A \preceq M_A$ we show that the initial states of M'_A and M_A also fulfill the requirements of the generalized mixed simulation relation.

First, for every ${s_a^0}' \in {S_A^0}'$, by the construction it holds that $\exists s_c^0 \in S_C^0$ s.t. $s_c^0 \in \gamma'({s_a^0}')$. Let $s_a^0 \in S_A$ be the superstate of ${s_a^0}'$, i.e., $\gamma'({s_a^0}') \subseteq \gamma(s_a^0)$. Then in particular $\exists s_c^0 \in \gamma(s_a^0)$ s.t. $s_c^0 \in \gamma(s_a^0)$. This implies that $s_a^0 \in S_A^0$ (by the construction of the exact GTS). Moreover, $({s_a^0}', s_a^0) \in H$ by the definition of H.

In the opposite direction, for every $s_a^0 \in S_A^0$, by the construction it holds that $\exists s_c^0 \in S_C^0$ s.t. $s_c^0 \in \gamma(s_a^0)$. Let ${s_a^0}' \in S'_A$ be the substate of s_a^0 s.t. $s_c^0 \in \gamma'({s_a^0}')$. Then, ${s_a^0}' \in {S_A^0}'$ (by the construction of the exact GTS). Moreover, $({s_a^0}', s_a^0) \in H$ by the definition of H. $\square$

Theorem 6.11 claims that for exact GTSs, refinement that is based on splitting abstract states is monotonic. This is true without the need to hold "copies" of the unrefined abstract states. Yet, as claimed before, constructing the exact GTS is not practical. Therefore, we suggest a compromise that fits well into the framework of abstraction-refinement. We show how to construct an initial abstract GTS and how to construct a refined abstract GTS (based on splitting abstract states). The construction is done in a way that is on the one hand computationally efficient and on the other hand maintains a monotonic refinement.

The basic idea is as follows. In each iteration of the abstraction-refinement we first construct an exact KMTS, including its may transitions and its (regular) must transitions. We transform the KMTS into a GTS by viewing each (regular) must transition (s, s') as a *hyper-transition* $(s, \{s'\})$ whose target set is the singleton set consisting of the target state of the regular transition. We then compute additional must *hyper-transitions* as described below.

Construction of an Initial Abstract Model M_0:
Given an initial set of abstract states S_0 and a concretization function γ_0:

1. construct an exact KMTS based on (S_0, γ_0) and transform it to a GTS.
2. If the transition relation of M_C is known to be total, then for every abstract state $s \in S_0$, add an outgoing must hyper-transition whose target set consists of the targets of all the outgoing may transitions of s.

Note that if the concrete transition relation is total, then the set of all the outgoing may transitions of a state indeed fulfills the $\forall\exists\exists$ condition and thus can be added as a must hyper-transition. This results from the property that every concrete transition is represented by some may transition. We call such must hyper-transitions *trivial*.

Construction of a Refined Model M_{i+1}:
Suppose that model checking of the abstract model M_i resulted in an indefinite result and refinement is needed. Let (S_{i+1}, γ_{i+1}) be the split of (S_i, γ_i), computed by some kind of a refinement mechanism. Construct M_{i+1} as follows.

1. construct an exact KMTS based on (S_{i+1}, γ_{i+1}) and transform it to a GTS.
2. for every must hyper-transition (including regular must transitions) $s_i R^+ A_i$ in M_i and for every state $s_{i+1} \in S_{i+1}$ that is a substate of $s_i \in S_i$, add to M_{i+1} the must hyper-transition $(s_{i+1}, \{s'_{i+1} \mid s'_{i+1}$ is a substate of $s'_i \in A_i$ and $s_{i+1} R^- s'_{i+1}\})$.

3. [optional] discard from M_{i+1} any must hyper-transition $s_{i+1}R^+A_{i+1}$ that is not *minimal*, which means that there is $s_{i+1}R^+A'_{i+1}$ in M_{i+1} where $A'_{i+1} \subset A_{i+1}$.

The purpose of step 2 above is to avoid the loss of information from the previous iteration, without paying an additional cost. To do so, we derive must hyper-transitions in M_{i+1} from must hyper-transitions in M_i, while avoiding the recomputation of the $\forall\exists\exists$ rule. Namely, if there is a must hyper-transition from s_i to A_i in M_i, then for every state s_{i+1} in M_{i+1} that is a substate of s_i we add an outgoing must hyper-transition to the set of all substates of states in A_i, excluding states to which s_{i+1} does not have a may transition. Clearly, given that $s_iR^+A_i$, we are guaranteed that the $\forall\exists\exists$ condition holds for the corresponding hyper-transitions in M_{i+1} as well. Note that this is not damaged by excluding from the target set states to which s_{i+1} does not have a may transition. This is because the lack of a may transition shows that the $\exists\exists$ condition does not hold between s_{i+1} and the relevant states. Therefore they cannot contribute to the satisfaction of the $\forall\exists\exists$ condition anyway and can be removed. By using this scheme, the construction of the GTS requires no additional computational cost, compared to the construction of a (regular) KMTS.

The purpose of step 3 is to reduce the GTS without sacrificing its precision. Note that the reduction done in this step can be performed *during* step 2.

Theorem 6.12. *Let M_C be a concrete Kripke structure and let $M_0, M_1, \ldots M_i, \ldots$ be the abstract GTSs constructed as described above. Then*
(1) for every $i \geq 0$: $M_C \preceq M_i$; and (2) for every $i \geq 0$: $M_{i+1} \preceq M_i$.

Proof. The proof is by induction on i. The base of the induction holds since M_0 is constructed as an abstract KMTS. We now prove the induction step.

We first prove (1). As in the exact GTS, the generalized mixed simulation H_{i+1} from M_C to M_{i+1} is given by $(s_c, s_{i+1}) \in H_{i+1}$ iff $s_c \in \gamma(s_{i+1})$. To prove that this is a generalized mixed simulation from M_C to M_{i+1} it suffices to refer to the must hyper-transitions (requirement 3), since the rest of the components of M_{i+1} are computed as in the exact KMTS (GTS) and thus we are guaranteed that the other requirements are fulfilled (see Theorem 6.8). We prove that requirement 3 holds for $(s_c, s_{i+1}) \in H_{i+1}$.

Suppose $s_{i+1}R^+_{i+1}A_{i+1}$. If the hyper-transition results from a regular must transition of the exact KMTS, then the requirement is fulfilled based on the properties of an abstract KMTS. Otherwise, the hyper-transition results from a must hyper-transition $s_iR^+_iA_i$ of M_i, where s_i is a superstate of s_{i+1}. This means that $A_{i+1} = \{s'_{i+1} \mid s'_{i+1}$ is a substate of $s'_i \in A_i$ and $s_{i+1}R^-s'_{i+1}\}$. We need to show that there is a state $s'_c \in S_C$ s.t. $s_cRs'_c$, and there is $s'_{i+1} \in A_{i+1}$ s.t. $(s'_c, s'_{i+1}) \in H_{i+1}$ (see the remark following Definition 6.6). Since s_i is a superstate of s_{i+1}, we have that $\gamma_{i+1}(s_{i+1}) \subseteq \gamma_i(s_i)$. Along with the definition of H_{i+1}, this implies that $s_c \in \gamma_{i+1}(s_{i+1}) \subseteq \gamma_i(s_i)$, and therefore $(s_c, s_i) \in H_i$. Thus, by the induction hypothesis, there exists a state $s'_c \in S_C$ s.t. $s_cRs'_c$, and there exists $s'_i \in A_i$ s.t. $(s'_c, s'_i) \in H_i$, meaning that $s'_c \in \gamma_i(s'_i)$. For the latter $s'_i \in A_i$, consider its substate s'_{i+1} s.t. $s'_c \in \gamma_{i+1}(s'_{i+1})$. Then, by the rules of the construction of the may transitions in an abstract KMTS, $s_{i+1}R^-s'_{i+1}$ (since $s_c \in \gamma_{i+1}(s_{i+1})$, $s'_c \in \gamma_{i+1}(s'_{i+1})$ and $s_cRs'_c$). This ensures that $s'_{i+1} \in A_{i+1}$. Moreover, $(s'_c, s'_{i+1}) \in H_{i+1}$ since $s'_c \in \gamma_{i+1}(s'_{i+1})$. This concludes the proof.

We now prove (2). The generalized mixed simulation H_{i+1} from M_{i+1} to M_i is given by $(s_{i+1}, s_i) \in H_{i+1}$ iff s_{i+1} is a substate of s_i. Again, to prove that this is a generalized mixed simulation from M_{i+1} to M_i it suffices to refer to the must hyper-transitions (requirement 3), since the rest of the components are constructed as in the exact KMTS (GTS), and thus the requirements for them are ensured by Theorem 6.11. We prove that requirement 3 holds for $(s_{i+1}, s_i) \in H_{i+1}$.

Suppose $s_i R_i^+ A_i$. Then the construction of M_{i+1} ensures that s_{i+1}, which is a substate of s_i, has a corresponding must hyper-transition $s_{i+1} R_{i+1}^+ A_{i+1}$ to $A_{i+1} = \{s'_{i+1} \mid s'_{i+1}$ is a substate of $s'_i \in A_i$ and $s_{i+1} R^- s'_{i+1}\}$. We need to show that $(A_{i+1}, A_i) \in H_{i+1}^{\forall\exists}$, i.e. that $\forall s'_{i+1} \in A_{i+1}\ \exists s'_i \in A_i : (s'_{i+1}, s'_i) \in H_{i+1}$. Consider some $s'_{i+1} \in A_{i+1}$. Then by the definition of A_{i+1}, we know that s'_{i+1} is a substate of some $s'_i \in A_i$, which ensures that $(s'_{i+1}, s'_i) \in H_{i+1}$. This concludes the proof of (2). $\square$

Theorem 6.12 first ensures that the construction of the initial and the refined GTSs described above yields abstract models which are greater by the generalized mixed simulation relation than the concrete model. Moreover, it ensures that although we do not use the exact GTSs, we still have a generalized mixed simulation relation between GTSs from different iterations in a monotonic fashion. This means that we do not lose information during the refinement and we get "closer" to the concrete model.

Note that the first part of the theorem holds even if in each step of the algorithm we do not base the construction on the *exact* KMTS. Moreover, the monotonicity of the must hyper-transitions is maintained as well. However, for the rest of the components monotonicity is not ensured if the construction does not use the exact KMTS.

Example 6.13. To demonstrate these ideas we return to the program P from Example 6.2 and see how the use of GTSs as described above affects it. The initial GTS M_0 is similar to the KMTS M from Figure 6.2(a), with two additional *trivial* must hyper-transitions from s_1 and from s_2 to $\{s_1, s_2\}$ (since in this case the concrete transition relation is known to be total). Yet, the truth value of $\varphi = \mu Z.((x \leq 0) \vee \Diamond Z)$ remains indefinite in this model. When we construct the refined model M_1 (based on the addition of the predicate $odd(x)$), we get a GTS that is similar to the KMTS M' from Figure 6.2(b), but M_1 also has additional must hyper-transitions. In particular, it has two trivial must hyper-transitions from both of its initial states to the set $\{s_{10}, s_{11}\}$. These must hyper-transitions are the refined version of the (regular) must transition from s_0 to s_1 in M_0: They exist because the initial states of M_1 are substates of the initial state s_0 of M_0 and the set $\{s_{10}, s_{11}\}$ consists of all the substates of s_1. Their existence in M_1 allows to verify φ, since due to them each of the initial states now has an outgoing must hyper-path in which all the paths reach s_{20} where $x \leq 0$.

Example 6.13 also demonstrates our advantage over [37] which stems from the fact that our refinement does not use "copies" of the unrefined abstract states, unlike [37]. This example shows that in our approach the *old* information (from the unrefined model) is expressed w.r.t. the *new* refined states. Consequently, the old information and the new information, for which refinement was needed, can be combined, resulting in a better precision.

To conclude this section and make the suggested ideas complete, it remains to provide (1) a model checking algorithm that evaluates μ-calculus formulas over GTSs, using the

$$\frac{s \vdash \Diamond\varphi}{t \vdash \varphi}\ \exists:\ sR^{+}\{t\} \text{ or } sR^{-}t \qquad \frac{s \vdash \Box\varphi}{t \vdash \varphi}\ \forall:\ sR^{+}\{t\} \text{ or } sR^{-}t$$

$$\frac{s \vdash \Diamond\varphi}{A \vdash \Diamond\varphi}\ \exists:\ sR^{+}A \qquad \frac{s \vdash \Box\varphi}{A \vdash \Box\varphi}\ \forall:\ sR^{+}A$$

$$\frac{A \vdash \Diamond\varphi}{t \vdash \varphi}\ \forall:\ t \in A \qquad \frac{A \vdash \Box\varphi}{t \vdash \varphi}\ \exists:\ t \in A$$

Figure 6.3: New model checking game rules for GTSs.

generalized 3-valued semantics; and (2) a suitable refinement mechanism to be applied when the model checking result is indefinite. Using these two components within the general framework suggested above, results in an actual abstraction-refinement framework where the refinement is monotonic.

Model Checking As a 3-valued model checking algorithm for GTSs we suggest a simple generalization of the game-based algorithms presented in Chapters 4 and 5. Namely, we modify the rules of the model checking game from Chapter 3 (see Figure 3.1) in configurations whose subformula is of the form $\Box\psi$ or $\Diamond\psi$ to account for the use of must hyper-transitions. The new rules appear in Figure 6.3.

First, we replace the role of regular must transitions by must hyper-transitions whose target set is a singleton and adapt the original rules accordingly (see first line of Figure 6.3). Moreover, we add rules that allow the players to use "real" must hyper-transitions: In any configuration of the form $(s, \Diamond\varphi)$ or $(s, \Box\varphi)$ where a player chooses to use a "real" must hyper-transition, his or her move is split into two steps. First, he or she chooses the desired must hyper-transition, $(s, A) \in S \times 2^S$ (second line of Figure 6.3). Then, the *other* player chooses a single state $t \in A$ and the play proceeds to the configuration (t, φ) (third line). Intuitively, since *all* the states in A *together* form the target of the must hyper-transition, whenever a player uses such a hyper-transition, he makes a statement about *all* the states in A. The opponent chooses *one* state from this set as a challenge to make it harder on the player, with the idea that if *some* state from A does not fulfill the expected property, the opponent will be able to choose it and entrap the player.

Lemma 3.6 is maintained (after a simple extension of Definition 3.5 to the new intermediate configurations) and as such the correctness of the game, described by Theorem 3.3, carries over to this case as well. The translation to a 3-valued parity game, as well as the algorithms for solving the game, remain the same.

Refinement As for the refinement mechanism, we use the algorithms suggested in Chapters 4 and 5 in order to find a failure vertex, analyze the failure and decide how to split the abstract states. To be convinced that no change is needed, one needs to notice that the refinement is based on may transitions only.

Moreover, the construction of a refined model M_{i+1} can be improved when this refinement mechanism is used. Namely, during the failure analysis it is possible to learn about additional must hyper-transitions that can be added to M_{i+1}. This is because if the cause for failure is a may transition $s_i R^- s'_i$ (in M_i) then the split is done by separating the set $conc_{must}$ of all the concrete states represented by s_i that have a corresponding outgoing transition from the rest of the states represented by s_i (see Section 4.3.2). In this case, we are guaranteed that after the split, the $\forall\exists\exists$ condition holds between the substate of s_i representing the concrete set $conc_{must}$ and the set containing all the substates of s'_i. Therefore, we can add such a must hyper-transition to M_{i+1} without additional computational cost.

Theorem 6.14. *For finite concrete models, iterating the suggested abstraction-refinement process is guaranteed to terminate with a definite answer.*

6.4 Concluding Remarks

In this chapter we investigate the non-monotonicity problem of KMTSs which arises during refinement. We identify the must transitions as the source of the problem and suggest *generalized KMTSs* (GTSs) as alternative abstract models for branching time logics such as the μ-calculus.

GTSs result in more precise abstract models in which more μ-calculus formulas can be proved or disproved. Yet, they suffer from a (potential) exponential blowup. In order to maintain the effectiveness of the abstraction-refinement technique while benefiting from the better precision of GTSs, we develop a new abstraction-refinement scheme. Namely, we adjust the 3-valued abstraction-refinement approach described in the previous chapters to the new monotonic framework. However, instead of constructing the *exact* GTS in each iteration of abstraction-refinement, we construct the abstract GTS in a more gradual fashion by taking into consideration the GTS from the previous iteration. We start with some initial GTS which consists of (mostly) ordinary transitions. Then, during refinement, when the abstract states are split, instead of computing all must hyper-transitions (as is done in the exact GTS), we"learn" must hyper-transitions from must transitions (and hyper-transitions) that existed in the previous iteration. Thus, in many cases we avoid the exponential blowup.

Chapter 7

More Precision at Less Cost

7.1 Introduction

In this chapter we investigate both the *precision* of abstract models for the μ-calculus, and the *efficiency* of their model checking. We provide a new definition of precision of abstract models, which measures precision w.r.t. the choice of the abstract states, independently of the formalism used to describe abstract models. We then show that GTSs are not yet satisfactory in terms of precision. We suggest how to overcome their imprecision by using may hyper-transitions. This results in a new class of models, for which we also suggest a construction of an abstract model which is most precise w.r.t. any choice of abstract states. We then suggest an efficient abstract model checking algorithm for the alternation-free μ-calculus that achieves the newly obtained maximal precision while avoiding the exponential blowup inherent by the use of hyper-transitions. To complete the picture, we incorporate our abstract model checking algorithm into an abstraction-refinement framework.

Abstract models are typically constructed based on some given abstraction. Recall that an abstraction consists of a set of abstract states S_A and a mapping (or concretization function) γ that defines the relation between abstract states and the concrete states that they represent. The rest of the components of the concrete model then also need to be lifted into the abstract world, in order to result in an abstract model. This can be done in various ways by using some *class* of abstract models.

For example, previous works used KMTSs [45, 38] that contain both must transitions and may transitions as abstract models. Other works used GTSs defined in Chapter 6, or their variants. In these models must hyper-transitions take the place of the (regular) must transitions. This is because must transitions were shown to behave badly in refinement in the sense of causing a loss of precision (see Chapter 6). They were also shown in [63, 27, 29] to be a source of incompleteness, in the sense that when limited to the use of must transitions, it is not always possible to construct a finite abstract model in which a property holds, even if it holds on the concrete model. As described in Chapter 6, must hyper-transitions prevent the loss of precision during refinement. They also result in a *complete* abstraction framework for the fragment of the μ-calculus defined with greatest fixpoints only [27, 29] ([27] also introduces fairness and hence achieves completeness for the full μ-calculus).

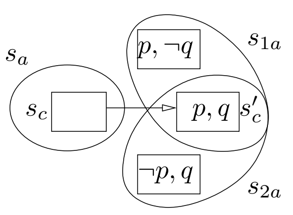

s_a
s_c
$p, \neg q$
s_{1a}
p, q
s'_c
$\neg p, q$
s_{2a}
Rectangles depict concrete states circled by the abstract states representing them.

Figure 7.1: Illustration of Example 7.1.

In this chapter, we are first interested in the *precision* of abstract models. Precision of an abstract model is measured by the extent to which it enables to verify or falsify formulas. Specifically, given an abstraction (S_A, γ), it is desirable to construct an abstract model over the states S_A in which as many formulas as possible have a definite value (true or false). With this purpose in mind, we address the allegedly non-problematic *may* transitions. We show that while being good enough for completeness purposes [27, 29], they are in fact a source of *imprecision*. This might sound surprising, yet the explanation is simple: when completeness is investigated, the choice of the abstraction (S_A, γ) is left open. On the other hand, when precision is investigated, one is interested in how precise the model is for a *given* abstraction.

In order to elaborate further on the imprecision problem we recall the definition of abstract models for the μ-calculus. To ensure logic preservation, may transitions in an abstract model have to be such that whenever there is a concrete transition from a concrete state s_c to a concrete state s'_c, then every abstract state that represents s_c has to have a may transition to some abstract state that represents s'_c. This is because the may transitions are used to over-approximate the concrete transitions. Now, consider the following example.

Example 7.1. Suppose that we are interested in verifying the formulas $\Box p$ ("all the successors satisfy p") and $\Box q$ ("all the successors satisfy q") in a concrete state s_c that has exactly one successor s'_c satisfying both p and q. Suppose further that we are given an abstraction in which s_c is represented by s_a, and no other concrete state is represented by s_a. Moreover suppose that s'_c is represented by two abstract states: s_{1a} that satisfies p but has an indefinite value on q, and s_{2a} that satisfies q but has an indefinite value on p. Figure 7.1 illustrates this setting. Then at least one of the transitions (s_a, s_{1a}) or (s_a, s_{2a}) has to be included as a may transition in the abstract model to ensure logic preservation. However, choosing the first will enable verification of $\Box p$, but not $\Box q$, choosing the second will enable the opposite, and including both transitions will prevent verification of both properties. In other words, no choice of a may transition relation will enable verification of both $\Box p$ and $\Box q$. In particular, none of them will enable to verify $\Box p \wedge \Box q$.

Intuitively, in order to achieve the desired precision in the above example one has to consider both may transitions, but each of them has to be considered separately. We therefore suggest a new class of models, called *Hyper Kripke Modal Transition Systems* (HTSs), in which may transitions are also replaced by hyper-transitions, with the meaning that each outgoing may hyper-transition of an abstract state s_a over-approximates *all* the concrete transitions of the states represented by s_a, but several different approximations (may hyper-transitions) can be used. Other possible solutions involve changing the abstract state space, for example by some kind of completion that improves the states precision (e.g. [25]). However, in this work we do not follow such solutions since we wish to "make the most" of the *given* abstract states.

Using HTSs as abstract models solves the problem demonstrated by Example 7.1, but one may wonder if there are other imprecision sources that HTSs do not address. To answer this question and justify the use of HTSs as abstract models we show how to construct, given *any* abstraction, an HTS which is as *precise* as the abstraction allows. We formalize this by introducing a new, simple, definition of precision which measures the precision w.r.t. the abstraction (S_A, γ) itself, independently of the class of abstract models used. This enables us to claim that the constructed HTS is as precise as possible, among all possible abstract models with a standard 3-valued semantics.

HTSs therefore settle the issue of precision, as they allow maximal precision. Yet, in terms of efficiency, their use only increases the problem which already exists in GTSs due to the must hyper-transitions: In general, the number of hyper-transitions might be exponential in the number of states in the abstract model. Thus, the need to handle hyper-transitions makes both the construction of an abstract model and its model checking computationally expensive.

This problem was already addressed in Chapter 6 w.r.t. must hyper-transitions. Recall that the solution there is to "learn" must hyper-transitions from must transitions (and hyper-transitions) that existed in the previous iteration of abstraction-refinement. Yet, this approach suffers from several disadvantages. First, it only works as a part of an abstraction-refinement loop. More importantly, the produced must hyper-transitions are not necessarily the ones that are needed in practice for a specific proof. Some of them might be redundant, as they are irrelevant for proving the desired property, whereas others which are needed to verify the desired property might not be produced, making the model not precise enough.

We wish to obtain efficiency without compromising the precision that an HTS enables to get. We achieve this goal for the alternation free fragment of the μ-calculus. The ability to do this results from the fact that the precise HTS is precise w.r.t. *every* μ-calculus formula, whereas we are only interested in *one* particular (alternation-free) formula. This can be exploited to save unnecessary efforts.

Suppose, for example, that we wish to check the formula $\Diamond p$ ("there is a successor that satisfies p") in an abstract state s_a, for which the number of outgoing must hyper-transitions in the precise HTS is exponential in the number of states. If we want the abstract model to be as precise as possible w.r.t every μ-calculus formula, we might need to consider all of the hyper-transitions (or at least the minimal ones). However, for the verification of $\Diamond p$ in s_a it suffices to consider a single must hyper-transition (underapproximation), in which all the target states satisfy p. In other words, w.r.t. the particular formula, an HTS that contains only the relevant must hyper-transition is as precise as the precise HTS. Similar reasoning applies to may hyper-transitions. The question is how to find these designated hyper-transitions and avoid the computation of the rest.

The key idea is to construct the HTS *during* the model checking, and thus avoid the (exponential) construction of the precise HTS. We use the model checking to guide the computation of hyper-transitions, by checking for the existence of hyper-transitions only when needed.

We obtain an automatic construction of an abstract model which is as precise as the precise HTS w.r.t. the property of interest, along with a new abstract model checking algorithm for the alternation-free μ-calculus with complexity $O(|S_A|^2 \times |\varphi|)$. This is comparable to the model checking complexity of the alternation free μ-calculus over models limited to ordinary transitions (recall that the number of ordinary transitions over $|S_A|$ states is $O(|S_A|^2)$), except that our algorithm also ensures maximal precision.

We emphasize that while may hyper-transitions are not always necessary for maximal precision, must hyper-transitions are in fact mandatory for completeness. This demonstrates the importance of such an algorithm, which handles both may and must hyper-transitions efficiently. Moreover, our approach can be beneficial even in cases where ordinary transitions suffice for the construction of a precise abstract model for a formula. This is because such constructions are usually expensive as they require finding best approximations of the concrete transitions (e.g. [26]). In our approach, instead of computing best approximations, the model checking algorithm wisely chooses candidates for which we perform the simpler task of checking if the *given* candidate is a correct approximation – not necessarily the best one.

To complete the discussion, we show how to use our abstract model checking within an abstraction-refinement framework, and show that the refinement has the desirable property of monotonicity, meaning that the precision of an abstract model never decreases as a result of refinement.

Finally, in Section 7.6, we consider concrete systems with multiple initial states, and show that similar imprecision and efficiency questions arise, and are settled by similar techniques.

7.1.1 Related Work

Precision of modal (or mixed) transition systems, with ordinary may and must transitions, is studied in [22, 26, 69]. They suggest constructions of such abstract models which are most precise among all models from this *specific* class. In Chapter 6 GTSs are considered. There, we suggest a construction of an abstract GTS (with must hyper-transitions) and show that it is most precise among all models produced by a *specific* construction method. In contrast to the above, this chapter defines a general notion of precision, which is independent not only of the construction method, but also of the class of abstract models.

A similar approach is taken in [43]. They refer to multi-valued concrete models and use an abstract semantics which is more general than the 3-valued semantics. They also define precision w.r.t. the abstraction itself, but then use (multi-valued) transition systems as abstract models, which causes a loss of precision. Our work, on the other hand, suggests a class of models that achieves maximal precision for the case of 2-valued concrete models. Moreover, [43] defines precision within the framework of abstract interpretation [24] and assumes that every set of concrete states has a unique most precise abstract state that describes it. We do not impose any restrictions on the abstraction and provide a simple, "stand alone", definition of precision.

The work of [29] also measures the precision of an abstract model by comparison to the precision of the abstraction. They define the precision of an abstraction (S_A, γ) in terms of a game over the concrete model. Their definition considers abstract states as precise in less cases than our definition. In particular, the abstract state s_a from Example 7.1 is not considered precise for $\Box p$ by their definition (when translating it to logic terms), although as demonstrated by Example 7.1, it *does* carry enough information to verify $\Box p$ in the (only) concrete state it represents. Using this stronger definition they show that the construction of an abstract model which is equivalent to the exact GTS (see Chapter 6) results in a precise abstract model. This is in contrast to our result that shows that GTSs do not allow maximal precision, since we measure the precision of a model compared to a more general definition of precision of an abstraction. As a consequence, when pursuing precision w.r.t. our definition, we get abstract models which are strictly more precise. They thus allow to verify and falsify more properties of the concrete model.

[37] refers to precision with a different motivation. They suggest how to define the abstraction (S_A, γ) after refinement in order to maintain precision of an abstract model after refinement. Thus, they measure precision only w.r.t. the precision before refinement and not independently.

A different approach to precision, pursued in [12, 38], uses a more precise 3-valued semantics, referred to as the *thorough semantics*. This semantics gives more definite answers than the standard 3-valued semantics, at the expense of increasing the complexity of model checking. Namely, the resulting model checking problem has the same complexity as satisfiability. We are interested in an effective framework, thus we use the standard 3-valued semantics, which is less precise, but enjoys a better model checking complexity. We note that the imprecision problem described in this chapter still exists even if the thorough semantics is used. Namely, the thorough semantics evaluates a formula in an abstract model depending on its value in all possible (consistent) concretizations of the abstract model. This is in general more precise than the standard (inductive) semantics which might implicitly consider inconsistent underlying concrete models. However, the described problem results from the imprecision of the abstract model itself, meaning that undesired concrete models are included as part of its "real" concretizations. Thus, even the thorough semantics, which considers only the real concretizations, does not help to overcome the imprecision.

May hyper-transitions resemble the de-focus operations of [27], just like must hyper-transitions resemble the focus operations. However, the focus and de-focus operations of [27] are used in the evaluation of $\vee$ and $\wedge$-formulas. We use the standard semantics for $\vee$ and $\wedge$, which does not depend on the underlying model. Instead, we use may and must hyper-transitions in the evaluation of $\Box$ and $\Diamond$-formulas, where the transitions of the underlying model need to be considered. In addition, in [27] the authors are interested in completeness and do not refer to the precision or model checking cost of the suggested class of models.

In terms of model checking, our approach in which we construct the abstract model during the model checking has some resemblance to the work of [67]. They perform reachability analysis, where they execute the concrete transitions, while storing abstract versions of the concrete states that are visited. Their approach is limited to falsification of safety properties, as they consider only an underapproximation of the concrete model. Our work, on the other hand, is suitable for any property expressed in the alternation free μ-calculus, and is based on a 3-valued setting which enables both verification and falsification.

7.2 Abstraction Framework

In order to allow a more thorough discussion of the precision of abstract models, which is not limited to the scope of a specific class of models, we define the 3-valued abstraction framework more generally.

Let M_C be a concrete Kripke structure with a set of concrete states S_C. Recall that an *abstraction* (S_A, γ) for S_C consists of a finite set of *abstract states* S_A and a total *concretization function* $\gamma : S_A \to 2^{S_C}$ that maps each abstract state to the (nonempty) set of concrete states it represents. Every $s_c \in S_C$ is represented by some $s_a \in S_A$.

The abstract states provide descriptions of the concrete states. The other components of the model M_C then also need to be lifted into the abstract world. Several classes of abstract models have been suggested for this purpose (examples are the classes of KMTSs and GTSs).

A *class of models* consists of some form of a transition system. It is accompanied with a *semantics* for the logic of interest, in our case the μ-calculus, over models from the class, and some *preservation relation* $\preceq$ between states that ensures preservation of the logic. An *abstract model* for M_C is then a model M_A from the class, over S_A, in which $(M_C, s_c) \preceq (M_A, s_a)$ whenever $s_c \in \gamma(s_a)$.[1]

We are particularly interested in classes of abstract models that use a 3-valued semantics, where the truth values are $\{\text{tt}, \text{ff}, \bot\}$, and where the preservation relation ensures preservation of both tt and ff. We refer to such classes of models as *3-valued classes*.

A 3-valued semantics was suggested for various classes of abstract models (e.g. [45, 39]. In order to allow a more uniform discussion, we define a *generic 3-valued semantics* that generalizes these definitions.

A 3-valued class defines, for each model M from the class, sets $l^M \in 2^S$, for every $l \in \mathit{Lit}$, and operators $\Box^M, \Diamond^M : 2^S \to 2^S$. These definitions are given in terms of the components of M (e.g. abstract transitions and labeling), with the requirements that l^M and $(\neg l)^M$ are disjoint and the operators $\Box^M$ and $\Diamond^M$ are monotone w.r.t. set inclusion. The 3-valued semantics for the class is then defined based on Kleene's 3-valued logic for $\wedge$ and $\vee$, and with the standard definition for fixpoints. Only the definition for formulas of the form $l \in \mathit{Lit}$, $\Box\psi$, and $\Diamond\psi$ depends on the particular class of M.

Definition 7.2 (Generic 3-Valued Semantics)**.** *Let M be a model from a 3-valued class. The* tt-set $[\![\varphi]\!]_{\text{tt}}^{M,\rho} \subseteq S$ *and the* ff-set $[\![\varphi]\!]_{\text{ff}}^{M,\rho} \subseteq S$ *of a μ-calculus formula φ over M and an environment $\rho : \mathcal{V} \to 2^S$ are defined inductively as follows.*

In the following definition the functionals $f = \lambda g.[\![\varphi]\!]_{\text{tt}}^{M,\rho[Z\mapsto g]}$ and $f = \lambda g.[\![\varphi]\!]_{\text{ff}}^{M,\rho[Z\mapsto g]}$ are elements of $2^S \to 2^S$ and $\text{lfp}(f)$ *and* $\text{gfp}(f)$ *denote their least and greatest fixpoints resp. These fixpoints exist according to [79] since the elements in 2^S form a complete*

[1]We consider concrete models that do not have a set of initial states. Therefore, we require state-wise preservation. In Section 7.6 we extend the discussion to models that have a set of initial states.

lattice under set inclusion ordering ($\subseteq$) and the functionals f are monotone w.r.t. this ordering for any Z, φ and S.

$$\begin{array}{lll@{\qquad}lll}
[\![l]\!]_{\mathrm{tt}}^{M,\rho} & := & l^M & [\![l]\!]_{\mathrm{ff}}^{M,\rho} & := & (\neg l)^M \\
[\![\Box\varphi]\!]_{\mathrm{tt}}^{M,\rho} & := & \Box^M([\![\varphi]\!]_{\mathrm{tt}}^{M,\rho}) & [\![\Box\varphi]\!]_{\mathrm{ff}}^{M,\rho} & := & \Diamond^M([\![\varphi]\!]_{\mathrm{ff}}^{M,\rho}) \\
[\![\Diamond\varphi]\!]_{\mathrm{tt}}^{M,\rho} & := & \Diamond^M([\![\varphi]\!]_{\mathrm{tt}}^{M,\rho}) & [\![\Diamond\varphi]\!]_{\mathrm{ff}}^{M,\rho} & := & \Box^M([\![\varphi]\!]_{\mathrm{ff}}^{M,\rho}) \\
[\![\varphi_1 \wedge \varphi_2]\!]_{\mathrm{tt}}^{M,\rho} & := & [\![\varphi_1]\!]_{\mathrm{tt}}^{M,\rho} \cap [\![\varphi_2]\!]_{\mathrm{tt}}^{M,\rho} & [\![\varphi_1 \wedge \varphi_2]\!]_{\mathrm{ff}}^{M,\rho} & := & [\![\varphi_1]\!]_{\mathrm{ff}}^{M,\rho} \cup [\![\varphi_2]\!]_{\mathrm{ff}}^{M,\rho} \\
[\![\varphi_1 \vee \varphi_2]\!]_{\mathrm{tt}}^{M,\rho} & := & [\![\varphi_1]\!]_{\mathrm{tt}}^{M,\rho} \cup [\![\varphi_2]\!]_{\mathrm{tt}}^{M,\rho} & [\![\varphi_1 \vee \varphi_2]\!]_{\mathrm{ff}}^{M,\rho} & := & [\![\varphi_1]\!]_{\mathrm{ff}}^{M,\rho} \cap [\![\varphi_2]\!]_{\mathrm{ff}}^{M,\rho} \\
[\![Z]\!]_{\mathrm{tt}}^{M,\rho} & := & \rho(Z) & [\![Z]\!]_{\mathrm{ff}}^{M,\rho} & := & \rho(Z) \\
[\![\mu Z.\varphi]\!]_{\mathrm{tt}}^{M,\rho} & := & \mathrm{lfp}(\lambda g.[\![\varphi]\!]_{\mathrm{tt}}^{M,\rho[Z\mapsto g]}) & [\![\mu Z.\varphi]\!]_{\mathrm{ff}}^{M,\rho} & := & \mathrm{gfp}(\lambda g.[\![\varphi]\!]_{\mathrm{ff}}^{M,\rho[Z\mapsto g]}) \\
[\![\nu Z.\varphi]\!]_{\mathrm{tt}}^{M,\rho} & := & \mathrm{gfp}(\lambda g.[\![\varphi]\!]_{\mathrm{tt}}^{M,\rho[Z\mapsto g]}) & [\![\nu Z.\varphi]\!]_{\mathrm{ff}}^{M,\rho} & := & \mathrm{lfp}(\lambda g.[\![\varphi]\!]_{\mathrm{ff}}^{M,\rho[Z\mapsto g]})
\end{array}$$

If φ is a closed formula, then $[\![\varphi]\!]_{\mathrm{tt}}^{M,\rho} = [\![\varphi]\!]_{\mathrm{tt}}^{M,\rho'}$, and $[\![\varphi]\!]_{\mathrm{ff}}^{M,\rho} = [\![\varphi]\!]_{\mathrm{ff}}^{M,\rho'}$, for any environments ρ, ρ'. Thus, when closed formulas are considered, we drop the environment from the semantic brackets.

If for every $\varphi \in \mathcal{L}_\mu$, $[\![\varphi]\!]_{\mathrm{tt}}^{M} \cap [\![\varphi]\!]_{\mathrm{ff}}^{M} = \emptyset$, then M is consistent. *The 3-valued semantics of $\varphi \in \mathcal{L}_\mu$ over M, denoted $[\![\varphi]\!]_3^M$, is then defined to be a mapping $S \to \{\mathrm{tt}, \mathrm{ff}, \bot\}$:*

$$[\![\varphi]\!]_3^M(s) = \begin{cases} \mathrm{tt}, & \textit{if } s \in [\![\varphi]\!]_{\mathrm{tt}}^M \\ \mathrm{ff}, & \textit{if } s \in [\![\varphi]\!]_{\mathrm{ff}}^M \\ \bot, & \textit{otherwise} \end{cases}$$

Remark 7.3. It is easy to verify that for consistent models, the 3-valued semantics $[\![\varphi]\!]_3^M$ can equivalently be defined as in Section 2.3, i.e., based on the lattice $(S \to \{\mathrm{tt}, \mathrm{ff}, \bot\}, \sqsubseteq)$, except that the semantics of formulas of the form $l \in Lit$, $\Box\psi$, or $\Diamond\psi$ is defined as follows.

$$\begin{array}{lll}
[\![l]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \mathrm{tt}, & \text{if } s \in l^M \\ \mathrm{ff}, & \text{if } s \in (\neg l)^M \\ \bot, & \text{otherwise} \end{cases} \\
[\![\Box\psi]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \mathrm{tt}, & \text{if } s \in \Box^M(\{s \mid [\![\varphi]\!]_3^{M,\rho}(s) = \mathrm{tt}\}) \\ \mathrm{ff}, & \text{if } s \in \Diamond^M(\{s \mid [\![\varphi]\!]_3^{M,\rho}(s) = \mathrm{ff}\}) \\ \bot, & \text{otherwise} \end{cases} \\
[\![\Diamond\psi]\!]_3^{M,\rho} & := & \lambda s. \begin{cases} \mathrm{tt}, & \text{if } s \in \Diamond^M(\{s \mid [\![\varphi]\!]_3^{M,\rho}(s) = \mathrm{tt}\}) \\ \mathrm{ff}, & \text{if } s \in \Box^M(\{s \mid [\![\varphi]\!]_3^{M,\rho}(s) = \mathrm{ff}\}) \\ \bot, & \text{otherwise} \end{cases}
\end{array}$$

Here, the environment ρ is a mapping $\mathcal{V} \to (S \to \{\mathrm{tt}, \mathrm{ff}, \bot\})$.

Note that if M is an abstract model, preservation of both tt and ff of $\mathcal{L}_\mu$ from M to the concrete model guarantees that M is consistent. From now on we restrict the discussion to abstract models, which ensures their consistency.

Example 7.4. An example of a 3-valued class of models is the class of GTSs with generalized mixed simulation as a relation that ensures logic preservation. For a GTS $M = (S, R^+, R^-, L)$, we define $l^M, \Box^M, \Diamond^M$ as follows. For every $l \in Lit$, $l^M = \{s \mid l \in$

$L(s)\}$. For every $U \subseteq S$: $\Box^M(U) = \{s \mid \forall t \in S,$ if $sR^- t$ then $t \in U\}$, and $\Diamond^M(U) = \{s \mid \exists A \subseteq S$ s.t. $sR^+ A$ and $A \subseteq U\}$. When integrated into Definition 7.2 along with Remark 7.3 this results in the 3-valued semantics of GTSs defined in Section 6.2.2.

Remark 7.5. The need to first separately define the tt-sets and the ff-sets of μ-calculus formulas rather than immediately defining the 3-valued semantics arises since in the general case the value of a formula in a state of a model from a 3-valued class can be both tt and ff, resulting in a 4-valued semantics.

When talking about KMTSs and GTSs, the requirement that the must (hyper) transitions are included in the may transitions ensures consistency and prevents such a scenario. More generally, this requirement ensures that if $U_1, U_2 \subseteq S$ are disjoint sets, then $\Box^M(U_1) \cap \Diamond^M(U_2) = \emptyset$, which ensures (by induction) that for every $\varphi \in \mathcal{L}_\mu$, $[\![\varphi]\!]^M_{\mathrm{tt}} \cap [\![\varphi]\!]^M_{\mathrm{ff}} = \emptyset$. Therefore, for KMTSs and GTSs the 3-valued semantics can immediately be defined as a mapping $S \to \{\mathrm{tt}, \mathrm{ff}, \bot\}$, without the need to first define the tt-sets and the ff-sets separately (as defined in Sections 2.3 and 6.2.2).

One could think of requiring an equivalent requirement from any 3-valued class in order to ensure consistency. However, such a requirement also restricts the expressiveness of the models. For example, when considering KMTSs, the constructions of [26] do not maintain this requirement, even though they are consistent. These constructions only ensure that if U_1 and U_2 represent disjoint sets of *concrete states* then $\Box^M(U_1) \cap \Diamond^M(U_2) = \emptyset$. This is a sufficient condition for consistency. Yet, since it involves the underlying concrete states, it cannot be used in the more general context.

Thus, we do not add such restrictions. This allows the consideration of more expressive classes of models, at the price of complicating the semantics and allowing the value of a formula in a state to be both tt and ff. However, as explained above, when considering an abstract model, consistency is ensured, and a 3-valued semantics is obtained.

7.3 Increasing Precision

Let M_C be a concrete Kripke structure. In this section we are interested in the *precision* of the abstract model constructed for M_C with a given abstraction (S_A, γ).

Specifically, in Section 6.2.2 we described GTSs as a class of abstract models, along with constructions of abstract models from this class. We now ask the following questions: (1) Do the constructions of GTSs from Section 6.2.2 produce the most precise abstract model that we can hope for, given an abstraction? and more fundamentally: (2) Does the use of GTSs *enable* to express the most precise abstract model?[2]

Of course, to answer these questions we first need to define what the most precise abstract model that we can hope for is, given an abstraction. We measure precision w.r.t. a 3-valued semantics. We therefore restrict the discussion to abstract models from 3-valued classes.

[2]In the discussion of GTSs in this chapter we omit the requirement that the must (hyper) transitions are included in the may transitions in order to allow more generality. Moreover, we do not consider a set of initial states.

7.3.1 Precision of Abstract Models

We wish to capture maximal precision within the boundaries of the inductive 3-valued semantics as defined in Definition 7.2. When using this semantics, the verification or falsification of any $\mathcal{L}_\mu$ formula over an abstract model M_A boils down to manipulations of l^{M_A}, $\Box^{M_A}(U_A)$, and $\Diamond^{M_A}(U_A)$ for various $l \in \mathit{Lit}$ and $U_A \subseteq S_A$. We therefore view a set $U_A \subseteq S_A$ as a new formula with the following semantics.

Definition 7.6. *Let (S_A, γ) be an abstraction for a concrete Kripke structure M_C, and let $U_A \subseteq S_A$. Then, for every concrete state s_c in M_C, $[\![U_A]\!]^{M_C}(s_c) = \mathrm{tt}$ iff $s_c \in \gamma(U_A)$, where $\gamma(U_A)$ stands for $\bigcup_{s_a \in U_A} \gamma(s_a)$.*

Moreover, for any abstract model M_A for M_C (from a 3-valued class), $[\![U_A]\!]^{M_A}_{\mathrm{tt}} = U_A$, meaning that $[\![U_A]\!]^{M_A}_{3}(s_a) = \mathrm{tt}$ iff $s_a \in U_A$.

We do not define the ff-sets of formulas of the form $U_A \subseteq S_A$ since they are not needed for our further developments. Moreover, the definition of the tt-sets of such formulas does not depend on an environment since they are considered closed formulas.

Using Definition 7.6 in conjunction with Definition 7.2, we get that $\Box^{M_A}(U_A) = [\![\Box U_A]\!]^{M_A}_{\mathrm{tt}}$, and $\Diamond^{M_A}(U_A) = [\![\Diamond U_A]\!]^{M_A}_{\mathrm{tt}}$. In addition, recall that $l^{M_A} = [\![l]\!]^{M_A}_{\mathrm{tt}}$. Since model checking boils down to manipulations of l^{M_A}, $\Box^{M_A}(U_A)$, and $\Diamond^{M_A}(U_A)$, this makes the tt-sets of formulas of the form l, $\Box U_A$, and $\Diamond U_A$ the building blocks of any model checking problem over M_A. As such, the precision of M_A is determined by its precision w.r.t. truth of such formulas.

In the spirit of [29] we first define the precision of an *abstraction* w.r.t. such formulas. This is the precision that a precise abstract *model* will then be expected to match.

Definition 7.7 (Precision of Abstractions)**.** *Given an abstraction (S_A, γ) for M_C and a state $s_a \in S_A$, we say that s_a* fulfills *$\varphi = l$, $\Box U_A$ or $\Diamond U_A$, for $l \in \mathit{Lit}$ and $U_A \subseteq S_A$, if $\forall s_c \in \gamma(s_a) : [\![\varphi]\!]^{M_C}(s_c) = \mathrm{tt}$.*

Note that this definition is independent of the class of abstract models, as it is meant to capture the precision of the abstraction itself, in terms of the information carried within the abstract states. For example, for the abstraction to reflect the fact that $\Box U_A$ holds in an abstract state s_a (meaning it holds in all the concrete states it represents), it has to be the case that *all* the concrete states in $\gamma(s_a)$ share the property that all of their outgoing (concrete) transitions are to $\gamma(U_A)$, which is the "description" of U_A in the concrete world.

Definition 7.8 (Precision of Models)**.** *An abstract model M_A for M_C (from some 3-valued class) is* precise *w.r.t. (S_A, γ) if for all $s_a \in S_A$, $l \in \mathit{Lit}$ and $U_A \subseteq S_A$: whenever s_a fulfills $\varphi = l$, $\Box U_A$ or $\Diamond U_A$, then $[\![\varphi]\!]^{M_A}_{3}(s_a) = \mathrm{tt}$.*

Thus whenever the information about l, $\Box U_A$, or $\Diamond U_A$ exists in the abstract states, a precise abstract model enables to see that. To formalize the generality of Definition 7.8, we extend Definition 7.7 to more complicated formulas and to falsification, following the 3-valued semantics. We then show that whenever an abstract model is precise w.r.t. truth of $l, \Box U_A, \Diamond U_A$, it is also precise w.r.t. any other formula.

Definition 7.9. *Let $\mathcal{A} = (S_A, \gamma)$ be an abstraction. We define an* abstract semantics *$[\![\varphi]\!]_3^{\mathcal{A}}$ by using the generic 3-valued semantics (see Definition 7.2) with the following definitions of $l^{\mathcal{A}} \in 2^{S_A}$, and $\Box^{\mathcal{A}}, \Diamond^{\mathcal{A}} : 2^{S_A} \to 2^{S_A}$. For $l \in Lit$: $l^{\mathcal{A}} = \{s_a \mid s_a$ fulfills $l\}$. For $U_A \subseteq S_A$: $\Box^{\mathcal{A}}(U_A) = \{s_a \mid s_a$ fulfills $\Box U_A\}$, and $\Diamond^{\mathcal{A}}(U_A) = \{s_a \mid s_a$ fulfills $\Diamond U_A\}$. We say that $s_a \in S_A$* enables verification (falsification) *of $\varphi \in \mathcal{L}_\mu$ if $[\![\varphi]\!]_3^{\mathcal{A}}(s_a) = \mathrm{tt}$ (ff).*

Recall that by Definition 7.2, $[\![\varphi]\!]_3^{\mathcal{A}}(s_a) = \mathrm{tt}$ if $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A}}$, and $[\![\varphi]\!]_3^{\mathcal{A}}(s_a) = \mathrm{ff}$ if $s_a \in [\![\varphi]\!]_{\mathrm{ff}}^{\mathcal{A}}$. The abstract semantics is well defined since whenever $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A}}$ (resp. $[\![\varphi]\!]_{\mathrm{ff}}^{\mathcal{A}}$), then $\forall s_c \in \gamma(s_a) : [\![\varphi]\!]^{M_C}(s_c) = \mathrm{tt}$ (resp. ff). This ensures that $[\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A}} \cap [\![\varphi]\!]_{\mathrm{ff}}^{\mathcal{A}} = \emptyset$.

For example, by this definition s_a enables verification of $\varphi = \Box\psi$ iff s_a fulfills $\Box U_A$ for some $U_A \subseteq S_A$ such that every $s'_a \in U_A$ enables verification of ψ.

Theorem 7.10. *Let M_A be an abstract model for M_C (from some 3-valued class) which is* precise *w.r.t. (S_A, γ). Then whenever $s_a \in S_A$ enables verification (falsification) of $\varphi \in \mathcal{L}_\mu$, then $[\![\varphi]\!]_3^{M_A}(s_a) = \mathrm{tt}$ (ff).*

Note that $[\![\varphi]\!]_3^{M_A}$ is well-defined since M_A is an abstract model for M_C, thus it is consistent.

Proof. We prove that if s_a enables verification of φ, i.e. $[\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A}}(s_a) = \mathrm{tt}$, then $[\![\varphi]\!]_3^{M_A}(s_a) = \mathrm{tt}$. The proof for falsification is implied since the 3-valued semantics ensures that $[\![\varphi]\!]_3^{M_A}(s_a) = \mathrm{ff}$ iff $[\![\neg\varphi]\!]_3^{M_A}(s_a) = \mathrm{tt}$, and similarly for the abstract semantics, where $\neg\varphi$ stands for the formula resulting by pushing the negation to the literals, while exchanging $\wedge$ with $\vee$, $\Box$ with $\Diamond$, and μ with ν. The proof is by induction on the structure of μ-calculus formulas. More specifically, we prove that for every μ-calculus formula φ and every environment ρ, if $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$, then $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{M_A,\rho}$. This implies that for a closed formula, if $[\![\varphi]\!]_3^{\mathcal{A}}(s_a) = \mathrm{tt}$, i.e. $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A}} = [\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$, then $s_a \in [\![\varphi]\!]_{\mathrm{tt}}^{M_A,\rho} = [\![\varphi]\!]_{\mathrm{tt}}^{M_A}$ and hence $[\![\varphi]\!]_3^{M_A}(s_a) = \mathrm{tt}$. The interesting cases are when $\varphi = l \in Lit$, $\Box\psi$, or $\Diamond\psi$. The remaining cases are immediate as both $[\![\varphi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$ and $[\![\varphi]\!]_{\mathrm{tt}}^{M_A,\rho}$ are defined according to the generic 3-valued semantics.

- If $\varphi = l \in Lit$ and $s_a \in [\![l]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$, then since $[\![l]\!]_{\mathrm{tt}}^{\mathcal{A},\rho} = l^{\mathcal{A}}$ and by the definition of $l^{\mathcal{A}}$ we conclude that s_a fulfills l. Thus by the definition of a precise model $s_a \in [\![l]\!]_{\mathrm{tt}}^{M_A} = [\![l]\!]_{\mathrm{tt}}^{M_A,\rho}$.

- Suppose $s_a \in [\![\Box\psi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$. This means that $s_a \in \Box^{\mathcal{A}}([\![\psi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho})$. By the definition of $\Box^{\mathcal{A}}$, this means that s_a fulfills $\Box U_A$ for $U_A = [\![\psi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$. By the induction hypothesis for ψ, for every such $s'_a \in U_A = [\![\psi]\!]_{\mathrm{tt}}^{\mathcal{A},\rho}$, we have that $s'_a \in [\![\psi]\!]_{\mathrm{tt}}^{M_A,\rho}$. Thus $U_A \subseteq [\![\psi]\!]_{\mathrm{tt}}^{M_A,\rho}$. Moreover, since s_a fulfills $\Box U_A$, then by the definition of a precise model $s_a \in [\![\Box U_A]\!]_{\mathrm{tt}}^{M_A}$, meaning that $s_a \in \Box^{M_A}([\![U_A]\!]_{\mathrm{tt}}^{M_A}) = \Box^{M_A}(U_A)$. Monotonicity of $\Box^{M_A}$ w.r.t. $\subseteq$ implies that $\Box^{M_A}(U_A) \subseteq \Box^{M_A}([\![\psi]\!]_{\mathrm{tt}}^{M_A,\rho})$, and therefore $s_a \in \Box^{M_A}([\![\psi]\!]_{\mathrm{tt}}^{M_A,\rho})$, thus by the 3-valued semantics $s_a \in [\![\Box\psi]\!]_{\mathrm{tt}}^{M_A,\rho}$ as well. The case of $\varphi = \Diamond\psi$ is similar.

□

The next theorem ensures that an abstract model which is precise w.r.t. the abstraction is also most precise when compared to other abstract models, provided that their class has the following property.

Definition 7.11. *A 3-valued class of models is* structural *if its definitions of* $\Box^M, \Diamond^M : 2^{S_A} \to 2^{S_A}$ *ensure that for every abstract model M_A from the class based on an abstraction (S_A, γ) for a concrete model M_C, and for every $U_A \subseteq S_A$, whenever $s_a \in \Box^M(U_A)$, then for every $s_c \in \gamma(s_a)$ all the concrete successors of s_c are in $\gamma(U_A)$. Similarly, whenever $s_a \in \Diamond^M(U_A)$, then every $s_c \in \gamma(s_a)$ has a successor in $\gamma(U_A)$.*

Note that for every $U_A \subseteq S_A$ which is equal to $[\![\varphi]\!]^M_{\text{tt}}$ for some $\varphi \in \mathcal{L}_\mu$, the conditions of Definition 7.11 are guaranteed to hold, since in this case $\Box^M(U_A) = [\![\Box\varphi]\!]^M_{\text{tt}}$, and similarly $\Diamond^M(U_A) = [\![\Diamond\varphi]\!]^M_{\text{tt}}$. Thus the conditions are implied by the preservation guarantee of the class. However, for a class to be structural, we require that these conditions hold for *every* $U_A \subseteq S_A$. Intuitively, for $\Box^M$ and $\Diamond^M$ to maintain such consistency with the concrete world, they have to be based on some (structural) abstract description of the concrete transitions in the abstract model. For example, KMTSs, GTSs and their variants are such classes.

Theorem 7.12. *Let M_A, M'_A be two abstract models for M_C (from possibly different 3-valued classes) based on an abstraction (S_A, γ). If M_A is* precise *w.r.t. (S_A, γ) and the class of M'_A is structural, then for every $s_a \in S_A$ and every $\varphi \in \mathcal{L}_\mu$: $[\![\varphi]\!]^{M'_A}_3(s_a) \neq \perp \Rightarrow [\![\varphi]\!]^{M_A}_3(s_a) = [\![\varphi]\!]^{M'_A}_3(s_a)$.*

Proof. Let M'_A be *some* abstract model as described in the theorem, and M_A a precise model w.r.t. (S_A, γ). We prove that for every $s_a \in S_A$, if $[\![\varphi]\!]^{M'_A}_3(s_a) = \text{tt}$, then $[\![\varphi]\!]^{M_A}_3(s_a) = \text{tt}$. The proof for falsification is implied since the 3-valued semantics ensures that $[\![\varphi]\!]^{M'_A}_3(s_a) = \text{ff}$ iff $[\![\neg\varphi]\!]^{M'_A}_3(s_a) = \text{tt}$ and similarly for M_A, where $\neg\varphi$ stands for the formula resulting by pushing the negation to the literals as in the proof of Theorem 7.10. The proof is by induction on the structure of μ-calculus formulas. More specifically, we prove that for every μ-calculus formula φ and every environment ρ, if $s_a \in [\![\varphi]\!]^{M'_A,\rho}_{\text{tt}}$, then $s_a \in [\![\varphi]\!]^{M_A,\rho}_{\text{tt}}$. This implies that for a closed formula, if $[\![\varphi]\!]^{M'_A}_{\text{tt}}(s_a) = \text{tt}$, i.e. $s_a \in [\![\varphi]\!]^{M'_A}_{\text{tt}} = [\![\varphi]\!]^{M'_A,\rho}_{\text{tt}}$, then $s_a \in [\![\varphi]\!]^{M_A,\rho}_{\text{tt}} = [\![\varphi]\!]^{M_A}_{\text{tt}}$ and hence $[\![\varphi]\!]^{M_A}_3(s_a) = \text{tt}$. As before, we present the interesting cases where $\varphi = l \in \mathit{Lit}$, $\Box\psi$, or $\Diamond\psi$. The remaining cases are immediate as both $[\![\varphi]\!]^{M'_A,\rho}_{\text{tt}}$ and $[\![\varphi]\!]^{M_A,\rho}_{\text{tt}}$ are defined with the generic 3-valued semantics.

- For $\varphi = l \in \mathit{Lit}$, if $s_a \in [\![l]\!]^{M'_A,\rho}_{\text{tt}} = [\![l]\!]^{M'_A}_{\text{tt}}$, then by the preservation guarantee of M'_A, we conclude that $\forall s_c \in \gamma(s_a)$, $s_c \in [\![l]\!]^{M_C}$ (i.e. $[\![l]\!]^{M_C}(s_c) = \text{tt}$), thus s_a fulfills l. Since M_A is precise w.r.t. (S_A, γ), we conclude that $s_a \in [\![l]\!]^{M_A}_{\text{tt}} = [\![l]\!]^{M_A,\rho}_{\text{tt}}$.

- Suppose $\varphi = \Box\psi$, and $s_a \in [\![\Box\psi]\!]^{M'_A,\rho}_{\text{tt}}$. Let $U_A = [\![\psi]\!]^{M_A,\rho}_{\text{tt}}$. To show that $s_a \in [\![\Box\psi]\!]^{M_A,\rho}_{\text{tt}}$, we need to show that $s_a \in \Box^{M_A}(U_A)$. Since $s_a \in [\![\Box\psi]\!]^{M'_A,\rho}_{\text{tt}}$, then by the 3-valued semantics, $s_a \in \Box^{M'_A}([\![\psi]\!]^{M'_A,\rho}_{\text{tt}})$. By the induction hypothesis, $[\![\psi]\!]^{M'_A,\rho}_{\text{tt}} \subseteq [\![\psi]\!]^{M_A,\rho}_{\text{tt}} = U_A$. Thus, by monotonicity of $\Box^{M'_A}$ w.r.t. $\subseteq$, we conclude that $\Box^{M'_A}(\llbracket\psi\rrbracket_{\text{tt}}^{M'_A,\rho}) \subseteq \Box^{M'_A}(U_A)$, thus $s_a \in \Box^{M'_A}(U_A)$. Since M'_A belongs to a structural class, this ensures us that for every $s_c \in \gamma(s_a)$ all the concrete successors of s_c are in $\gamma(U_A)$, and thus belong to $\llbracket U_A\rrbracket^{M_C}$. Thus $\forall s_c \in \gamma(s_a) : s_c \in \llbracket\Box U_A\rrbracket^{M_C}$ (i.e., $\llbracket\Box U_A\rrbracket^{M_C}(s_c) = \text{tt}$). Thus by definition s_a fulfills $\Box U_A$. Since M_A is precise w.r.t. (S_A, γ), this ensures that $s_a \in \llbracket\Box U_A\rrbracket_{\text{tt}}^{M_A}$. Thus by the 3-valued semantics $s_a \in \Box^{M_A}(U_A) = \Box^{M_A}(\llbracket\psi\rrbracket_{\text{tt}}^{M_A,\rho})$, and $s_a \in \llbracket\Box\psi\rrbracket_{\text{tt}}^{M_A,\rho}$. The proof for $\varphi = \Diamond\psi$ is similar.

$\Box$

Now, equipped with formal definitions of precision, we go back to our questions about the precision of GTSs.

Theorem 7.13. *If the abstraction (S_A, γ) partitions the concrete states, i.e. for each $s_a, s'_a \in S_A : \gamma(s_a) \cap \gamma(s'_a) = \emptyset$, then the exact GTS from section 6.2.2 is precise w.r.t. (S_A, γ).*

Proof. Let M_A denote the exact GTS from section 6.2.2.

- Suppose that $s_a \in S_A$ fulfills $l \in \mathit{Lit}$. This means that $\forall s_c \in \gamma(s_a) : \llbracket l\rrbracket^{M_C}(s_c) = \text{tt}$ (i.e. $s_c \in \llbracket l\rrbracket^{M_C}$), and by the concrete semantics this implies that $\forall s_c \in \gamma(s_a) : l \in L_C(s_c)$. Therefore by the construction of the exact GTS, $l \in L_A(s_a)$ and hence $s_a \in \{s \mid l \in L_A(s)\} = l^{M_A} = \llbracket l\rrbracket_{\text{tt}}^{M_A}$.

- Suppose that $s_a \in S_A$ fulfills $\varphi = \Box U_A$. Thus, $\forall s_c \in \gamma(s_a) : \llbracket\Box U_A\rrbracket^{M_C}(s_c) = \text{tt}$ (i.e. $s_c \in \llbracket\Box U_A\rrbracket^{M_C}$). This means that $\forall s_c \in \gamma(s_a)\ \forall s'_c$, if $s_c R s'_c$ then $s'_c \in \llbracket U_A\rrbracket^{M_C}$. In other words, $\forall s_c \in \gamma(s_a)\ \forall s'_c$, if $s_c R s'_c$ then $s'_c \in \gamma(U_A)$ (1). Now, consider an outgoing may transition of s_a to some s'_a in M_A. It was computed based on the $\exists\exists$ condition, meaning that $\exists s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(s'_a)$ s.t. $s_c R s'_c$. By (1), this also ensures that $s'_c \in \gamma(U_A)$. Thus there exists $s''_a \in U_A$ such that $s'_c \in \gamma(s''_a)$. Since we have a partition, it implies that $s'_a = s''_a$ (since also $s'_c \in \gamma(s'_a)$). Thus $s'_a \in U_A$ and as such $s'_a \in \llbracket U_A\rrbracket_{\text{tt}}^{M_A}$. As this is true for every outgoing may transition of s_a, we conclude that $s_a \in \Box^{M_A}(\llbracket U_A\rrbracket_{\text{tt}}^{M_A}) = \llbracket\Box U_A\rrbracket_{\text{tt}}^{M_A}$.

- Suppose that $s_a \in S_A$ fulfills $\varphi = \Diamond U_A$. Thus, $\forall s_c \in \gamma(s_a) : \llbracket\Diamond U_A\rrbracket^{M_C}(s_c) = \text{tt}$ (i.e., $s_c \in \llbracket\Diamond U_A\rrbracket^{M_C}$). This means that $\forall s_c \in \gamma(s_a)\ \exists s'_c$ such that $s_c R s'_c$ and $s'_c \in \llbracket U_A\rrbracket^{M_C}$, i.e. $s'_c \in \gamma(U_A)$. In other words, $\forall s_c \in \gamma(s_a)\ \exists s'_c \in \gamma(U_A)$ such that $s_c R s'_c$. Thus, by the construction there exists a must hyper-transition in M_A from s_a to U_A, where all the states belong to $\llbracket U_A\rrbracket_{\text{tt}}^{M_A}$ (by definition). Thus $s_a \in \Diamond^{M_A}(U_A) = \Diamond^{M_A}(\llbracket U_A\rrbracket_{\text{tt}}^{M_A}) = \llbracket\Diamond U_A\rrbracket_{\text{tt}}^{M_A}$.

$\Box$

However, in many cases it might be desirable to gather the concrete states into non-disjoint sets, as this can reduce the size of the abstract state space that enables verification or falsification of the desired property. We show that in this general setting, the answer to both questions is “no”.

7.3.2 May Transitions as a Source of Imprecision

As demonstrated by Example 7.1, when the given abstract states do not represent disjoint sets of concrete states, the may transitions can become a source of imprecision. In this example there is *no* abstract GTS for M_C over S_A that will enable verification of both $\Box p$ and $\Box q$ in s_a. This is while the abstraction *does* enable verification of both $\Box p$ and $\Box q$ in s_a (see Definition 7.9). Thus, none of the possible GTSs is precise w.r.t. the given abstraction. As this example does not involve must hyper-transitions, the same conclusion holds even if we relax the requirement that the must hyper-transitions are included in the may transitions.

Theorem 7.14. *GTSs do not always suffice for the construction of a precise abstract model w.r.t. a given abstraction, even if the requirement that the must hyper-transitions are included in the may transitions is omitted.*

We emphasize that this imprecision is not limited to a certain construction. Indeed, the construction of the exact GTS from Section 6.2.2 is simplistic, as it might introduce redundancy in the may transitions (for example, in Example 7.1 both may transitions would be included). Yet, Theorem 7.14 holds even for optimized constructions that avoid redundant may transitions (e.g. in the style of [26]).

It can be shown that the imprecision results from the may transitions and not from the other components of the GTS. This is because whenever the abstraction enables verification of $l \in Lit$ or $\Diamond U_A$, so does the exact GTS, which implies that the labeling and the must hyper-transitions (used for verification of such formulas) are precise enough.

More than that, analyzing Example 7.1 shows that the imprecision arises when there is no "best" choice of may transitions, in which case one needs to consider *all* of their (incomparable) possibilities to achieve maximal precision. Unfortunately, a GTS does not enable to do that. We therefore suggest to model the may transitions as hyper-transitions as well, with the meaning that each may hyper-transition $(s_a, A_a) \in S_A \times 2^{S_A}$ provides *some* overapproximation of *all* the outgoing transitions of the concrete states represented by s_a.

7.3.3 Hyper Kripke Modal Transition Systems

This brings us to the new class of abstract models that we suggest to be used in order to obtain maximal precision.

Definition 7.15. *A* Hyper Kripke Modal Transition System *(***HTS***) is a tuple $M = (S, R^+, R^-, L)$, where S, L, R^+ are defined as before, and $R^- \subseteq S \times 2^S$.*

In particular, the target sets of the may and must hyper-transitions of an HTS might be empty.

3-Valued Semantics for HTSs To adapt the 3-valued semantics of $\mathcal{L}_\mu$ for HTSs we redefine $\Box^M$. The definitions of l^M and $\Diamond^M$ are the same as for GTSs. For every $U \subseteq S$: $\Box^M(U) = \{s \mid \exists A \subseteq S \text{ s.t. } sR^-A \text{ and } \forall t \in A : t \in U\}$. This changes the definition for

$\Box\psi$ in a consistent HTS to:

$$[\![\Box\psi]\!]_3^M(s) = \begin{cases} \text{tt}, & \text{if } \exists A \subseteq S \text{ s.t. } sR^-A \text{ and} \\ & \qquad \forall t \in A : [\![\psi]\!]_3^M(t) = \text{tt} \\ \text{ff}, & \text{if } \exists A \subseteq S \text{ s.t. } sR^+A \text{ and} \\ & \qquad \forall t \in A : [\![\psi]\!]_3^M(t) = \text{ff} \\ \bot, & \text{otherwise} \end{cases}$$

and dually for $[\![\Diamond\psi]\!]_3^M(s)$ when exchanging tt with ff.

Thus, in order to evaluate a $\Box\psi$ formula to tt, instead of requiring that *all* the may transitions of s are to states that satisfy ψ, we now require that there *exists* a may hyper-transition of s such that *all* the states within the target set satisfy ψ. This is justified by the fact that in an abstract HTS, *each* may hyper-transition of s (as opposed to *all* the may transitions of s together in an abstract GTS) will over-approximate *all* the concrete transitions leaving the concrete states represented by s.

Note that an HTS might be inconsistent. For example, a state s of an HTS M might have both a may hyper-transition to $[\![l]\!]_{\text{tt}}^M = \{s' \mid l \in L(s')\}$ and a must hyper-transition to $[\![l]\!]_{\text{ff}}^M = \{s' \mid \neg l \in L(s')\}$. This means that $s \in [\![\Box l]\!]_{\text{tt}}^M \cap [\![\Box l]\!]_{\text{ff}}^M$. Yet, we are interested in abstract HTSs which are always consistent.

A GTS, and thus also a Kripke structure, can be viewed as an HTS, where every state has exactly *one* outgoing may hyper-transition, whose target set consists of the target states of *all* of its (ordinary) may transitions. This encoding preserves the logical semantics of the models. Note that we allow the target of a may hyper-transition to be an empty set, in case the (may) transition relation of the Kripke structure or the GTS is not total. In particular, a may hyper-transition whose target set is empty causes the value of a formula of the form $\Box\psi$ to be tt in the source state of the hyper-transition. This coincides with the case of a state that has no outgoing (may) transitions in a Kripke structure or a GTS. Preservation of $\mathcal{L}_\mu$ between HTSs (and in particular between an HTS and a Kripke structure) is then guaranteed by the following relation.

Definition 7.16 (Hyper Mixed Simulation)**.** *Let $M_1 = (S_1, R_1^+, R_1^-, L_1)$ and $M_2 = (S_2, R_2^+, R_2^-, L_2)$ be two HTSs, both defined over AP. We say that $H \subseteq S_1 \times S_2$ is a* hyper mixed simulation *from M_1 to M_2 if $(s_1, s_2) \in H$ implies the following:*

1. *$L_2(s_2) \subseteq L_1(s_1)$.*
2. *if $s_2R_2^-A_2$, then there is some $A_1 \subseteq S_1$ s.t. $s_1R_1^-A_1$ and $(A_1, A_2) \in H^{\forall\exists}$.*
3. *if $s_2R_2^+A_2$, then there is some $A_1 \subseteq S_1$ s.t. $s_1R_1^+A_1$ and $(A_1, A_2) \in H^{\forall\exists}$,*

where as before: $(A_1, A_2) \in H^{\forall\exists} \Leftrightarrow \forall s_1' \in A_1\ \exists s_2' \in A_2 : (s_1', s_2') \in H$.

If there is a hyper mixed simulation H s.t. $(s_1, s_2) \in H$, then $(M_1, s_1) \preceq (M_2, s_2)$.

Thus, the requirements of a hyper mixed simulation are the same as those of a generalized mixed simulation (see Definition 6.6), except that requirement 2 is replaced. Namely, instead of requiring that for each may transition of M_1, there exists a corresponding may transition in M_2 such that the target states satisfy $(s_1', s_2') \in H$, i.e. s_2' over-approximates s_1', we now require that for each may hyper-transition of M_2 there exists a corresponding may hyper-transition in M_1 (note that the indices are swapped), such that the target sets satisfy $(A_1, A_2) \in H$, i.e. A_2 over-approximates A_1.

Intuitively, there can be less may hyper-transitions in M_2 but each one has to over-approximate *some* hyper-transition in M_1. Thus, if some may hyper-transition was used to verify $\Box\psi$ in M_2, then the may hyper-transition that it over-approximates can be used to verify it in M_1. Note that a may hyper-transition of M_1 that has no representation in M_2 can only cause formulas with a definite value in M_1 to be indefinite in M_2 and not vice versa.

When applying Definition 7.16 to a (concrete) Kripke structure M_C and an (abstract) HTS M_A, requirements 2 and 3 simplify as follows. For $(s_c, s_a) \in H$,

2. if $s_a R^- A_a$, then for every s'_c s.t. $s_c R s'_c$, there is $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$.
3. if $s_a R^+_A A_a$, then there is some s'_c s.t. $s_c R s'_c$ and there is $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$.

The simplification of requirement 3 is the same as in the case of generalized mixed simulation (see Chapter 6). As for requirement 2, recall that it requires that for $s_a R^- A_a$ there exists $s_c R^- A_c$ such that $(A_c, A_a) \in H^{\forall\exists}$. Yet, when viewing a Kripke structure M_C as an HTS, every state $s_c \in S_C$ has exactly one outgoing may hyper-transition $s_c R^- A_c$ where A_c consists of all the target states of the ordinary transitions of s_c, i.e., $A_c = \{s'_c \mid s_c R s'_c\}$. Therefore requirement 2 narrows down to this may hyper-transition, and $(A_c, A_a) \in H^{\forall\exists}$ translates into: for every $s'_c \in A_c$, i.e., for every s'_c s.t. $s_c R s'_c$, there exists $s'_a \in A_a$ s.t. $(s'_c, s'_a) \in H$.

Theorem 7.17. *For HTSs M_1 and M_2 with states s_1 and s_2 resp., if $(M_1, s_1) \preceq (M_2, s_2)$ then for every $\varphi \in \mathcal{L}_\mu$: $s_2 \in [\![\varphi]\!]^{M_2}_{\mathrm{tt}} \Rightarrow s_1 \in [\![\varphi]\!]^{M_1}_{\mathrm{tt}}$, and $s_2 \in [\![\varphi]\!]^{M_2}_{\mathrm{ff}} \Rightarrow s_1 \in [\![\varphi]\!]^{M_1}_{\mathrm{ff}}$.*

Proof. The proof is obtained by induction on the structure of μ-calculus formulas, similarly to the proof of Theorem 6.7. The only changes occur in cases where the semantics was changed, i.e. where may hyper-transitions are used instead of (ordinary) may transitions.

- Suppose $s_2 \in [\![\Box\psi]\!]^{M_2,\rho}_{\mathrm{tt}}$. Then by the definition of the semantics there exists a may hyper-transition from s_2 to A_2 such that for each $s'_2 \in A_2$: $s'_2 \in [\![\psi]\!]^{M_2,\rho}_{\mathrm{tt}}$. Moreover, since $(s_1, s_2) \in H$, we know that there exists A_1 such that s_1 has a may hyper-transition to A_1 and $(A_1, A_2) \in H^{\forall\exists}$, meaning that $\forall s'_1 \in A_1\ \exists s'_2 \in A_2 : (s'_1, s'_2) \in H$. Let s'_1 be such a state in A_1 and s'_2 the corresponding state from A_2. Since $s'_2 \in A_2$, we know that $s'_2 \in [\![\psi]\!]^{M_2,\rho}_{\mathrm{tt}}$. By the induction hypothesis, this implies that $s'_1 \in [\![\psi]\!]^{M_1,\rho}_{\mathrm{tt}}$. That is, $\forall s'_1 \in A_1$: $s'_1 \in [\![\psi]\!]^{M_1,\rho}_{\mathrm{tt}}$. Thus $s_1 \in [\![\Box\psi]\!]^{M_1,\rho}_{\mathrm{tt}}$. The treatment of the case where $s_2 \in [\![\Diamond\psi]\!]^{M_2,\rho}_{\mathrm{ff}}$ is dual.

□

In particular, if Theorem 7.17 is applied on a concrete Kripke structure M_C and an abstract HTS M_A for it (meaning M_A is consistent), then whenever $s_c \in \gamma(s_a)$, we have that $[\![\varphi]\!]^{M_A}_3(s_a) \neq \perp \Rightarrow [\![\varphi]\!]^{M_C}(s_c) = [\![\varphi]\!]^{M_A}_3(s_a)$.

Construction of an Abstract HTS Let $M_C = (S_C, R, L_C)$ be a (concrete) Kripke structure. Given an abstraction (S_A, γ) for it, an abstract model in the form of an HTS $M_A = (S_A, R^+, R^-, L_A)$, can be constructed as a GTS (see Section 6.2.2) with the exception that R^- now consists of hyper-transitions, constructed as follows. A may hyper-transition $s_a R^- A_a$ exists only if a [$\forall\forall\exists$] condition holds:

$$\forall s_c \in \gamma(s_a)\ \forall s_c'\ [\ s_c R s_c' \Rightarrow \exists s_a' \in A_a \text{ s.t. } s_c' \in \gamma(s_a')\ ]$$

That is, every outgoing may hyper-transition of s_a over-approximates *all* the concrete transitions of the states represented by s_a. In other words, each of the target sets of the outgoing may hyper-transitions of s_a over-approximates *all* the targets of the concrete transitions leaving the concrete states represented by s_a. An example of a "legal" may hyper-transition that satisfies the $\forall\forall\exists$ condition is (s_a, A_a) for every $s_a \in S_A$ and $A_a = \{s_a' \mid \exists s_c \in \gamma(s_a)\ \exists s_c' \in \gamma(s_a') \text{ s.t. } s_c R s_c'\}$. Note that the "only if" allows to include less hyper-transitions than allowed by the rule. The following theorem formalizes the correctness of the construction.

Theorem 7.18. *Let M_C be a concrete Kripke structure over S_C, and let M_A be an HTS computed as described above based on an abstraction (S_A, γ) for S_C. Then whenever $s_c \in \gamma(s_a)$ then $(M_C, s_c) \preceq (M_A, s_a)$.*

Proof. We show that $H \subseteq S_C \times S_A$ defined by $(s_c, s_a) \in H$ iff $s_c \in \gamma(s_a)$ is a hyper mixed simulation. Let $s_c \in \gamma(s_a)$. Requirements 1 and 3 regarding the labeling and the must hyper-transitions are fulfilled as in a GTS. We now refer to requirement 2. Let $A_a \subseteq S_A$ be such that $s_a R^- A_a$. By the remark following Definition 7.16, we need to show that for every s_c' s.t. $s_c R s_c'$, there is $s_a' \in A_a$ s.t. $(s_c', s_a') \in H$. Since $s_a R^- A_a$, this means (by the construction) that $\forall s_c \in \gamma(s_a)\ \forall s_c'\ [\ s_c R s_c' \Rightarrow \exists s_a' \in A_a \text{ s.t. } s_c' \in \gamma(s_a')\ ]$. In particular, for our s_c, we have that $\forall s_c'\ [\ s_c R s_c' \Rightarrow \exists s_a' \in A_a \text{ s.t. } s_c' \in \gamma(s_a')\ ]$. By the definition of H, $s_c' \in \gamma(s_a')$ implies that $(s_c', s_a') \in H$. Thus, the requirement holds. $\square$

For example, to verify $\Box p$ and $\Box q$ in Example 7.1, we include $(s_a, \{s_{1a}\})$ and $(s_a, \{s_{2a}\})$ as may hyper-transitions. Note that both of these hyper-transitions satisfy the $\forall\forall\exists$ condition, which ensures that each of them over-approximates *all* the concrete transitions of the concrete state represented by s_a (in this case there is only one such concrete transition). In addition, the labeling function defines $L_A(s_{1a}) = \{p\}$, and $L_A(s_{2a}) = \{q\}$. Now, the may hyper-transition $(s_a, \{s_{1a}\})$ enables to verify $\Box p$. Similarly, the may hyper-transition $(s_a, \{s_{2a}\})$ enables to verify $\Box q$. Thus, $\Box p \wedge \Box q$ is verified.

Exact HTS If the "only if" in the definition of may hyper-transitions is replaced by "iff", the may hyper-transitions are *exact*. If all components are exact, we get the *exact HTS*.

Theorem 7.19. *Let M_C be a Kripke structure and M_A^E the* exact HTS *computed as described above based on an abstraction (S_A, γ). Then M_A^E is precise w.r.t. (S_A, γ).*

Proof. The cases where $s_a \in S_A$ fulfills $l \in \mathit{Lit}$ or $\Diamond U_A$ are exactly as in the proof of theorem 7.13 (note that the proof of these cases did not rely on the fact that we had a partition of the concrete states). We refer to the remaining case, which is different.

- Suppose that $s_a \in S_A$ fulfills $\varphi = \Box U_A$. This means that $\forall s_c \in \gamma(s_a) : [\![\Box U_A]\!]^{M_C}(s_c) =$ tt (i.e. $s_c \in [\![\Box U_A]\!]^{M_C}$). In other words, $\forall s_c \in \gamma(s_a)\ \forall s'_c$, if $s_c R s'_c$ then $s'_c \in [\![U_A]\!]^{M_C}$, or equivalently $\forall s_c \in \gamma(s_a)\ \forall s'_c$, if $s_c R s'_c$ then $s'_c \in \gamma(U_A)$, meaning that, $\forall s_c \in \gamma(s_a)\ \forall s'_c$ [$s_c R s'_c \Rightarrow \exists s'_a \in U_a$ s.t. $s'_c \in \gamma(s'_a)$]. Thus, by the construction of the exact HTS there exists a may hyper-transition in M_A from s_a to U_A. In addition, all the (abstract) states in U_A belong to $[\![U_A]\!]^{M_A}_{\text{tt}}$ (by definition), thus by the 3-valued semantics over HTS $s_a \in \Box^{M_A}(U_A) = \Box^{M_A}([\![U_A]\!]^{M_A}_{\text{tt}}) = [\![\Box U_A]\!]^{M_A}_{\text{tt}}$.

□

Optimization As suggested in Chapter 6 for must hyper-transitions, an HTS can be reduced without damaging its precision by discarding may and must hyper-transitions (s_a, A_a) that are not *minimal*, meaning that there is another hyper-transition (s_a, A'_a) of the same type where $A'_a \subset A_a$. In particular, Theorem 7.19 still holds after this optimization is applied.

For example, in the exact HTS constructed for Example 7.1, R^- (and also R^+) includes in addition to the hyper-transitions $(s_a, \{s_{1a}\})$ and $(s_a, \{s_{2a}\})$, the hyper-transition $(s_a, \{s_{1a}, s_{2a}\})$, which is not minimal. Thus the same precision is achieved when it is omitted. The same reasoning applies to other non-minimal hyper-transitions which are included in R^- and R^+, leaving us with $R^+ = R^- = \{(s_a, \{s_{1a}\}), (s_a, \{s_{2a}\})\}$.

Note that in an HTS, both the 3-valued semantics and the preservation relation (hyper mixed simulation) treat may and must hyper-transitions in the same way, rather than dually. Still, may hyper-transitions and must hyper-transitions have different roles in an abstract model: the first provides an overapproximation of the concrete transitions, and the latter provides an underapproximation for them. This difference is captured by the fact that when viewing a Kripke structure as an HTS, the may and must hyper-transitions are defined differently. Namely, each concrete transition is considered a must hyper-transition, whereas all the concrete transitions together form a single may hyper-transition. As such, the may and must hyper-transitions of an abstract HTS, which is related to the concrete Kripke structure by a hyper mixed simulation, are each required to satisfy different rules w.r.t. the concrete transitions. This is demonstrated by the construction of an abstract HTS, where the may and must hyper-transitions are defined differently.

7.4 Decreasing the Model Checking Cost

Using the exact HTS as an abstract model ensures maximal precision w.r.t. the given abstraction. Yet, it involves an exponential blowup (even with the suggested optimization). In this section we suggest an efficient model checking for the alternation-free μ-calculus, which remains quadratic in the number of abstract states, and yet produces a result which is *as precise as possible* w.r.t. a specific property.

In the remainder of this section we restrict the discussion to the alternation free fragment of the μ-calculus. Let M_C be a concrete Kripke structure and $\varphi \in \mathcal{L}^0_\mu$ a formula that we wish to check in some state s_c of M_C. Moreover, suppose that we are given a (finite) abstraction (S_A, γ). All the abstract states that represent s_c are

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

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

$$\frac{s \vdash \Diamond\psi}{t \vdash \psi} :\ s\widetilde{R}t \qquad \frac{s \vdash \Box\psi}{t \vdash \psi} :\ s\widetilde{R}t$$

Figure 7.2: Rules for the construction of the model checking graph.

candidates to enable verification or falsification of φ in s_c. We therefore refer to them as *designated* states. Our purpose is to evaluate φ in all these designated abstract states in the exact HTS M_A^E.

Our algorithm is based on a generalization of the game-based model checking suggested in [74] for CTL over KMTSs (with ordinary may and must transitions). We no longer formulate the problem as a game, but we use a graph that resembles the game graph of the model checking game as an underlying structure for our algorithm. We refer to this structure as the *model checking graph*, or in short *MC-graph*.

Model Checking Graph (MC-Graph) The MC-graph resembles the underlying graph of the model checking game for the μ-calculus presented in Section 3.2. It presents all the information "relevant" for the model checking: Every vertex in the graph is labeled by $s_a \vdash \psi$, where s_a is an abstract state and ψ is a subformula of φ, indicating that the value of ψ in s_a is relevant for determining the model checking result. The outgoing edges of a vertex $s_a \vdash \psi$ can be seen as defining "subgoals" for the goal of checking ψ in s_a.

Formally, let $\varphi \in \mathcal{L}_\mu^0$ be a formula, S_A a set of states, $S^d \subseteq S_A$ a set of designated states in which we want to evaluate φ, and $\widetilde{R} \subseteq S_A \times S_A$ a transition relation. $\widetilde{R}$ is meant to provide a basic description of the possible transitions between states (we will soon see how it is obtained). The *MC-graph* G w.r.t. φ, S_A, S^d and $\widetilde{R}$ is a graph (V, E) with a set of vertices $V \subseteq S_A \times \mathit{Sub}(\varphi)$ and a set of edges $E \subseteq V \times V$, defined as follows. The *initial vertices* $V_0 \subseteq V$ consist of $S^d \times \{\varphi\}$. The (rest of the) vertices and the edges are defined by the rules of Figure 7.2, with the meaning that whenever $v \in V$ is of the form of the upper part of the rule, then the result in the lower part of the rule is also a vertex $v' \in V$ and $(v, v') \in E$.

The rules in Figure 7.2 resemble those given in Figure 3.1 for the 3-valued model checking game over KMTSs, except that only one transition relation $\widetilde{R}$ is used (instead of both must and may transition relations).

The vertices of G are classified as $\wedge$, $\vee$, $\Box$, $\Diamond$, or literal vertices, based on their subformulas. Vertices whose subformulas are of the form Z or $\eta Z.\psi$ are *deterministic* – they have exactly one son. Vertices that have no outgoing edges are called *terminal vertices*.

Since φ is an alternation-free formula, each strongly connected component (SCC) in G which is non-trivial, i.e. has at least one edge, contains exactly one free fixpoint variable $Z \in \mathcal{V}$, called a *witness*. If $fp(Z) = \mu Z.\psi$, then Z is a μ-witness. Otherwise it is a ν-witness.

Coloring Algorithm To determine the model checking result, a coloring algorithm is applied on the MC-graph with the purpose of labeling each vertex $v = s_a \vdash \psi$ in it by T, F, ? depending on the value of ψ in the state s_a in M_A^E, based on the 3-valued semantics.

We start with some background on the coloring algorithm of [74]. There, the algorithm processes the graph of the model checking game in a bottom-up manner by iterating two phases: In the sons-coloring phase, a vertex is colored based on the colors of its sons by rules which reflect the 3-valued semantics of the logic. In the witness-coloring phase a special procedure is applied to handle cycles (non trivial SCCs) in the graph.

The witness-coloring phase analyzes vertices that remained uncolored after iterating the rules of the sons-coloring phase. Such vertices are necessarily a part of a non-trivial SCC which has a witness[3]. Depending on the witness, one of the definite colors (T or F) is ruled out for the remaining uncolored vertices, yet another phase is needed to decide between ? and the remaining definite color. For example, a μ-witness rules out the T-color, as infinite paths cannot contribute to satisfaction of a μ (least fixpoint) formula. Thus, for an uncolored vertex v in such an SCC it remains to be checked if the condition for coloring v by F, which depends on v's type, can still hold for it, and if not color it ?. This is done similarly to the sons-coloring phase, except that the rules are now aimed at checking that v has no potential to be colored F. The remaining vertices are colored F. A ν-witness, on the other hand, rules out the F-color, thus vertices that remain uncolored in such an SCC are colored T or ?.

As for our algorithm, for the sake of the explanation, suppose first that we construct the MC-graph based on M_A^E (of course, eventually the point will be to avoid the construction of M_A^E). The transition relation $\widetilde{R}$ will then simply be the set $\widetilde{R}^E = \{(s_a, s'_a) \mid s'_a \in A_a \text{ and } (s_a R^- A_a \text{ or } s_a R^+ A_a)\}$, where R^- and R^+ are the transition relations of M_A^E. That is, $\widetilde{R}^E$ contains all the (ordinary) transitions that participate in some hyper-transition in M_A^E. In this case we also define may and must hyper-sons in G: If $v = s \vdash \heartsuit\psi \in V$ for $\heartsuit \in \{\Box, \Diamond\}$ and sR^-A (sR^+A), then the set of vertices $B = A \times \{\psi\} \subseteq V$ is a *may (must) hyper-son* of v.

The coloring can be extended to handle hyper-sons in the same way that the 3-valued semantics is extended to handle hyper-transitions. For example, a $\Box$-vertex will be colored by F iff it has a must hyper-son whose vertices are all colored by F. It will be colored by T iff it has a may hyper-son whose vertices are all colored by T. Otherwise it will be colored ?. Dually for a $\Diamond$-vertex. Thus, the coloring algorithm can be seen, in a sense, as exhaustively trying to find the justification for coloring each vertex. Yet, instead of considering *all* the hyper-sons and checking if any of them justifies coloring the vertex, we suggest to use the information gathered so far in the bottom-up coloring to perform this check wisely.

For example, to color a $\Box$-vertex v by F, it suffices to check, whenever some son of

[3]In [74], the model checking game is designed for the logic CTL. In particular, a witness there is defined differently. We adapt the presentation to the alternation-free fragment of the μ-calculus.

v gets colored by F, if all of v's currently F-colored sons comprise a must hyper-son (i.e., their underlying states fulfill the $\forall\exists\exists$ rule). This is because must hyper-sons whose vertices are not all colored F will not justify coloring v by F, and thus need not be checked. Moreover, if some subset of the F-colored sons of v comprises a must hyper-son, then so does the full set. Similarly, to conclude that v should not be colored F (as is done in the witness-coloring phase), it suffices to check that v's currently F-colored sons along with the uncolored sons (if exist) do not form a must hyper-son. If they do not, then the same holds for any of their subsets, and clearly other sets of vertices cannot form a must hyper-son whose vertices are all colored F. This means that v has no potential to have a must hyper-son whose vertices are all colored by F, and it is safe to conclude that it cannot be colored F. Thus, checking these candidates is as informative as checking *all* of the possible must hyper-sons. Similar reasoning applies to may hyper-sons.

This leads us to the following algorithm, where M_A^E is *not* constructed in advance.

7.4.1 Optimized Abstract Model Checking

Let M_C be a concrete model, $s_c \in S_C$ a concrete state, $\varphi \in \mathcal{L}_\mu^0$ a formula that we wish to check in s_c, and (S_A, γ) an abstraction. The algorithm is as follows.

MC-Graph Construction Construct a *partial* HTS $\widetilde{M_A} = (S_A, \widetilde{R}, L_A)$, where L_A is defined as in the exact HTS, and $\widetilde{R} \subseteq S_A \times S_A$ is defined by $\widetilde{R} = \{(s_a, s_a') \mid \exists s_c \in \gamma(s_a)\ \exists s_c' \in \gamma(s_a') \text{ s.t. } s_c R s_c'\}$. This ensures that $\widetilde{R} \supseteq \widetilde{R}^E$. Construct the MC-graph G based on φ, S_A, $\widetilde{R}$ as above, and $S^d = \{s_a \mid s_c \in \gamma(s_a)\}$.

Partition The MC-graph G is partitioned into Maximal Strongly Connected Components (MSCCs), denoted Q_i's, and a (total) order $\leq$ is determined on them, such that for every $v \in Q_i$ and $v' \in Q_j$, $(v, v') \in E$ only if $Q_j \leq Q_i$. Such an order exists because the MSCCs form a directed acyclic graph.

Coloring The following two phases are performed repeatedly until all vertices are colored.

1. *Sons-coloring phase.* Apply the following rules until none is applicable.
    - A terminal vertex $s_a \vdash l$ where $l \in Lit$ is colored T if $l \in L_A(s_a)$, F if $\neg l \in L_A(s_a)$, and ? otherwise.
    - A terminal vertex of the form $s_a \vdash \Box\psi$ ($s_a \vdash \Diamond\psi$) is colored T (F).
    - An $\wedge$-vertex ($\vee$-vertex) is colored by:
        - $T(F)$ if both its sons are colored $T(F)$.
        - $F(T)$ if it has a son that is colored $F(T)$.
        - ? if it has a son that is colored ? and the other is colored $\neq F(T)$.
    - A deterministic vertex is colored as its (only) son.
    - A non-terminal $\Box$-vertex ($\Diamond$-vertex) is colored by:

  - $T(F)$ if its currently $T(F)$-colored sons form a may hyper-son.
  - $F(T)$ if its currently $F(T)$-colored sons form a must hyper-son.
  - ? if all of its sons are colored, yet none of the above holds.

2. *Witness-coloring phase.* If there are still uncolored vertices, let Q_i be the smallest MSCC w.r.t. $\leq$ that is not yet fully colored. Q_i is necessarily a non-trivial MSCC that has exactly one witness. Its uncolored vertices are colored according to the witness. For a μ-witness:

   (a) Repeatedly color ? each vertex in Q_i satisfying one of the following.
      - An $\wedge$-vertex ($\vee$-vertex) that both (at least one) of its sons are colored $\neq F$.
      - A deterministic vertex whose son is colored ?.
      - A $\Box$-vertex ($\Diamond$-vertex) whose F-colored sons along with its remaining uncolored sons do not form a must (may) hyper-son.

   (b) Color the remaining vertices in Q_i by F.

   The case where the witness is of type ν is dual, when exchanging F with T, $\wedge$ with $\vee$, and $\Box$ with $\Diamond$.

In each phase of the coloring, the rules will initially be checked once for every uncolored vertex, and later will only be checked when one of the sons of the vertex gets colored by an appropriate color. Several optimizations can be used. For example, during phase 1 it is possible to color a $\Box$-vertex ($\Diamond$-vertex) by ? before all of its sons are colored by checking that all of its $T(F)$-colored sons along with its uncolored sons do not form a may hyper-son, and in addition all of its $F(T)$-colored sons along with its uncolored sons do not form a must hyper-son. Note that if one of these conditions holds at one time then it will remain valid, thus it need not be checked again.

Remark 7.20. *Checking if a set B of vertices forms a may or must hyper-son of a $\Box$-vertex or a $\Diamond$-vertex v is performed by checking the $\forall\forall\exists$ or the $\forall\exists\exists$ condition (resp.) between the underlying states of the vertex v and the set of vertices B.*

The following theorem formalizes the correctness of the algorithm, by relating the colors of vertices in the MC-graph to truth values of the corresponding formulas in the corresponding states. To refer to formulas in the MC-graph which are not closed, we use the following notation. For a (possibly not closed) alternation free formula φ_1, φ_1^* denotes the result of replacing every free occurrence of $Z \in \mathcal{V}$ in φ_1 by $fp(Z)$. φ_1^* is always a closed formula.

Theorem 7.21. *Let M_A^E denote the exact HTS for M_C w.r.t. (S_A, γ) and let $\varphi \in \mathcal{L}_\mu^0$. Furthermore, let G be the MC-graph produced by the algorithm. Then for every $v = s_a \vdash \varphi_1 \in G$ the following holds:*

1. $[\![\varphi_1^*]\!]_3^{M_A^E}(s_a) = \mathrm{tt}$ *iff* $v = s_a \vdash \varphi_1$ *is colored by* T.

2. $[\![\varphi_1^*]\!]_3^{M_A^E}(s_a) = \mathrm{ff}$ *iff* $v = s_a \vdash \varphi_1$ *is colored by* F.

3. $[\![\varphi_1^*]\!]_3^{M_A^E}(s_a) = \perp$ *iff* $v = s_a \vdash \varphi_1$ *is colored by* ?.

In the proof of the theorem we use a variation of the Knaster-Tarski theorem (see Theorem 2.2) which holds for HTSs. Instead of using approximants as defined in Chapter 2, we formulate the theorem using unwindings of fixpoint formulas. Namely, for a formula $\eta Z.\psi$, we denote by ψ^i the unwinding of the fixpoint formula i times. Formally,

$$\psi^0 = \begin{cases} false, & \text{if } \eta = \mu \\ true, & \text{if } \eta = \nu \end{cases}$$

and $\psi^{i+1} = \psi[Z := \psi^i]$. For any HTS M over S, we define $[\![true]\!]_{\text{tt}}^M = [\![false]\!]_{\text{ff}}^M = S$ and $[\![false]\!]_{\text{tt}}^M = [\![true]\!]_{\text{ff}}^M = \emptyset$. Then, for an HTS with a finite set of states S, and for a closed formula $\eta Z.\psi$, there exists $i \in \mathbb{N}$ such that $[\![\eta Z.\psi]\!]_{\text{tt}}^M = [\![\psi^i]\!]_{\text{tt}}^M$. Similarly, there exists $i \in \mathbb{N}$ such that $[\![\eta Z.\psi]\!]_{\text{ff}}^M = [\![\psi^i]\!]_{\text{ff}}^M$. We now return to the proof of Theorem 7.21.

Proof of Theorem 7.21 (sketch). The proof is by induction on the run of the coloring algorithm. For any vertex which is colored within the sons-coloring phase the correctness follows directly from the 3-valued semantics, combined with the fact that if *all* the T-colored sons of a vertex v do not comprise a must (may) hyper-son, then v does not have a must (may) hyper-son whose vertices are all colored T. Similarly for F. Thus it is sufficient to check if the selected candidates comprise hyper-sons, rather than considering all the possible subsets of sons, as described before.

As for vertices which are colored in the witness coloring phase, the proof consists of several steps. We demonstrate the idea of the proof for a Q_i with a witness Z of type μ (the proof for the case of a ν-witness is similar).

In this case vertices are either colored by ? (in phase 2a) or F (in phase 2b). The first step is thus to show that the remaining uncolored vertices in Q_i at the beginning of this phase should indeed not be colored T. Let $v = s_a \vdash \varphi_1$ be a vertex in Q_i. We show that if $[\![\varphi_1^*]\!]_3^{M_A^E}(s_a) = \text{tt}$, i.e. v should be colored T, then v must have already been colored T in the sons-coloring phase. Thus none of the uncolored vertices should be colored T.

Suppose that $fp(Z) = \mu Z.\psi$. Since the abstract state space is finite, there exists i such that $[\![\mu Z.\psi]\!]_{\text{tt}}^{M_A^E} = [\![\psi^i]\!]_{\text{tt}}^{M_A^E}$, where ψ^i denotes the unwinding of the fixpoint formula i times. Let φ_1 and s_a be such that $[\![\varphi_1^*]\!]_3^{M_A^E}(s_a) = \text{tt}$, i.e. $s_a \in [\![\varphi_1^*]\!]_{\text{tt}}^{M_A^E}$. By the definition of φ_1^*, $[\![\varphi_1^*]\!]_{\text{tt}}^{M_A^E} = [\![\varphi_1[Z := \mu Z.\psi]]\!]_{\text{tt}}^{M_A^E} = [\![\varphi_1]\!]_{\text{tt}}^{M_A^E, \rho[Z \mapsto [\![\mu Z.\psi]\!]_{\text{tt}}^{M_A^E}]} = [\![\varphi_1]\!]_{\text{tt}}^{M_A^E, \rho[Z \mapsto [\![\psi^i]\!]_{\text{tt}}^{M_A^E}]} = [\![\varphi_1[Z := \psi^i]]\!]_{\text{tt}}^{M_A^E}$. Note that $\varphi_1[Z := \psi^i]$ is fixpoint-free, and it does not contain any free variable. Thus, one can follow the inductive definition of the 3-valued semantics for $\wedge$, $\vee$, $\Box$ and $\Diamond$ and construct a finite tree over pairs of states and formulas that explains why $s_a \in [\![\varphi_1[Z := \psi^i]]\!]_{\text{tt}}^{M_A^E}$. The root of the tree is $(s_a, \varphi_1[Z := \psi^i])$. All the pairs (s'_a, φ') in the proof tree will be such that $s'_a \in [\![\varphi']\!]_{\text{tt}}^{M_A^E}$. For example, for a pair of the form $(s'_a, \varphi'_1 \wedge \varphi'_2)$, both (s'_a, φ'_1) and (s'_a, φ'_2) will be included as sons in the proof tree. For a pair $(s'_a, \Box\varphi')$ some set $A'_a \times \{\varphi'\}$ such that $s'_a R^- A'_a$ is a may hyper-transition in M_A^E and $A'_a \subseteq [\![\varphi']\!]_{\text{tt}}^{M_A^E}$ will be included. The property that all the pairs (s'_a, φ') in the tree are such that $s'_a \in [\![\varphi']\!]_{\text{tt}}^{M_A^E}$ ensures that ψ^0 will not be included in the tree, as $[\![\psi^0]\!]_{\mathrm{tt}}^{M_A^E} = [\![false]\!]_{\mathrm{tt}}^{M_A^E} = \emptyset$. In addition, since no free variables or fixpoints exist in the tree, the subformulas become strictly shorter along paths of the tree. These two properties ensure that every path in the tree eventually reaches a formula that does not contain ψ^j as a subformula for any $j \geq 0$. These will be the leaves of the tree, which makes the tree finite (this results from the fact that we have explicitly unwound the fixpoint).

We now map every formula in the proof tree back to the original formula that produced it (by replacing ψ^j by Z), i.e. if $\varphi' = \varphi[Z := \psi^j]$, then we define $\sigma(\varphi') = \varphi$. This defines a mapping $\tilde{\sigma}$ from the vertices of the proof tree to the vertices of the MC-graph: $\tilde{\sigma}(s'_a, \varphi') = s'_a \vdash \sigma(\varphi')$. Since the leaves of the proof tree do not contain formulas of the form ψ^j for any $j \geq 0$, we are guaranteed that all the leaves of the proof tree are mapped to vertices in the MC-graph that do not contain Z, thus they belong to smaller Q_j's. This is the crucial observation as it means that these vertices were already colored by the time the witness-coloring phase is applied on Q_i and by the induction hypothesis their coloring is correct, i.e. they are colored T. This provides the basis for an inductive argument (on the depth of the proof tree) that shows that for every vertex (s'_a, φ') in the proof tree, the corresponding vertex $\tilde{\sigma}(s'_a, \varphi')$ in the MC-graph could be colored T in the sons-coloring phase. The induction step follows by a case analysis on the type of subformulas, and results from the relation between the 3-valued semantics and the rules of the coloring in the sons-coloring phase. Since the sons-coloring phase is iterated as long as some rule is applicable, the corresponding vertices must have been indeed already colored T. For example, consider some pair $(s'_a, \Box\varphi')$ in the proof tree for which the tree contains as an explanation the pairs $A'_a \times \{\varphi'\}$ for some may hyper-transition $s'_a R^- A'_a$ such that $A'_a \subseteq [\![\varphi']\!]_{\mathrm{tt}}^{M_A^E}$. Then first by the definition of $\widetilde{R}$, and since $\sigma(\Box\varphi') = \Box\sigma(\varphi')$, all of the vertices $A'_a \times \{\sigma(\varphi')\}$ to which $A'_a \times \{\varphi'\}$ are mapped by $\tilde{\sigma}$ are sons of the vertex $s'_a \vdash \sigma(\Box\varphi')$ that corresponds to $(s'_a, \Box\varphi')$ in the MC-graph. By the induction hypothesis, all of these sons get colored T in the sons-coloring phase. Thus, at latest when the last of them gets colored T, then the algorithm finds out that the set of currently T-colored sons of $s'_a \vdash \sigma(\Box\varphi')$, which is a superset of $A'_a \times \{\sigma(\varphi')\}$, is a may hyper-son of $s'_a \vdash \sigma(\Box\varphi') = \tilde{\sigma}(s'_a \vdash \Box\varphi')$, and therefore colors it T during the sons-coloring phase. In particular, for the root of the proof tree, $(s_a, \varphi_1[Z := \psi^i])$, we have that $\tilde{\sigma}(s_a, \varphi_1[Z := \psi^i]) = s_a \vdash \varphi_1$, which ensures that $v = s_a \vdash \varphi_1$ already got colored T in the sons-coloring phase.

Now, for vertices which are colored in phase 2a, a similar analysis as in the sons-coloring phase, following the 3-valued semantics, combined with the fact that if *all* the F-colored sons along with the uncolored sons of a vertex v do not comprise a must (may) hyper-son then v does not have a must (may) hyper-son whose vertices are all colored F, shows that the vertices colored in phase 2a should not be colored F. Together with the previous argument saying that they should not be colored T, this ensures the correctness of their coloring by ?.

To complete the proof it remains to show that the vertices $s_a \vdash \varphi_1$ which are colored in phase 2b should indeed be colored F. Alternatively, it suffices to show that every vertex $s_a \vdash \varphi_1$ that should *not* be colored F, i.e. $[\![\varphi_1^*]\!]_3(s_a) \neq \mathrm{ff}$, is indeed already colored by a different color (T or ?) when this phase is reached. Again, since the state space is finite, there exists i such that $[\![\mu Z.\psi]\!]_{\mathrm{ff}}^{M_A^E} = [\![\psi^i]\!]_{\mathrm{ff}}^{M_A^E}$, where ψ^i denotes the unwinding of the fixpoint formula i times. In particular, $[\![\varphi_1^*]\!]_{\mathrm{ff}}^{M_A^E} = [\![\varphi_1[Z := \psi^i]]\!]_{\mathrm{ff}}^{M_A^E}$. The proof again uses a construction of a proof tree, except that now this is a proof tree that explains why $s_a \notin [\![\varphi_1[Z := \psi^i]]\!]_{\mathrm{ff}}^{M_A^E}$. Here again the crucial observation is that ψ^0 cannot be part of the proof tree, since $[\![\psi^0]\!]_{\mathrm{ff}}^{M_A^E} = [\![false]\!]_{\mathrm{ff}}^{M_A^E} = S_A$, whereas all the vertices (s'_a, φ') in the proof tree are such that $s'_a \notin [\![\varphi']\!]_{\mathrm{ff}}^{M_A^E}$. Thus all the leaves of the proof tree are mapped to vertices in smaller Q_i's, which are already colored correctly (i.e. $\neq F$) before phase 2b and by induction on the depth of the proof tree so are the rest of the vertices of the MC-graph which are mapped to internal vertices of the proof tree. For example, for a pair $(s'_a, \Box\varphi')$ the proof tree contains as an explanation a set $A'_a \times \{\varphi'\}$ such that A'_a contains at least one state s^*_a such that $s^*_a \notin [\![\varphi']\!]_{\mathrm{ff}}^{M_A^E}$ from every must hyper-transition $s'_a R^+ A^*_a$ in M_A^E. Thus, at latest in phase 2a, after the last of the vertices $s^*_a \vdash \sigma(\varphi')$ for $s^*_a \in A'_a$ gets colored $\neq F$ (this happens by the induction hypothesis), it holds that the set B of F-colored sons of $s'_a \vdash \sigma(\Box\varphi')$ along with its uncolored sons does not form a must hyper-son, since for every must hyper-transition of M_A^E at least one target state comprises a vertex which belongs to $A'_a \times \{\sigma(\varphi')\}$, and is thus colored $\neq F$ at this point, and does not belong to B. Thus, $s'_a \vdash \sigma(\Box\varphi')$ gets colored ?. $\Box$

Thus, for all the vertices in the MC-graph, the coloring is as precise as model checking with M_A^E, even though M_A^E is *not* constructed by the algorithm. Note that if φ_1 is closed then $\varphi_1^* = \varphi_1$. Thus, for a vertex $v = s_a \vdash \varphi_1$ whose formula is closed the theorem immediately implies that the color of v in G matches the truth value of φ_1 in the state s_a of M_A^E. In particular, this is true for $V_0 = S^d \times \{\varphi\}$. Therefore, by the choice of S^d, we are guaranteed that whenever the abstraction is precise enough, i.e., whenever one of the designated states enables verification or falsification of φ, at least one initial vertex will be colored by a definite color T or F, in which case by Theorems 7.21 and 7.17, $[\![\varphi]\!]^{M_C}(s_c) = \mathrm{tt}$ or ff respectively. Note, that it is impossible that some initial vertex will be colored T and another will be colored F. If all the initial vertices in the MC-graph are colored ?, then the result is indefinite.

Remark 7.22. *By considering the underlying hyper-transitions of hyper-sons computed by the algorithm, the final MC-graph induces an abstract HTS for M_C which is as precise as the exact HTS w.r.t. φ.*

Complexity During all applications of the sons-coloring phase, the $\forall\exists\exists$ and the $\forall\forall\exists$ conditions are checked at most $|S_A|$ times for each vertex, as each vertex has at most $|S_A|$ sons, and between checks the set of candidates to comprise a hyper-son is monotonically increasing. Similar analysis holds for phase 2a, with the difference that the sets of candidates to comprise a hyper-son are monotonically decreasing. As the number of vertices in the MC-graph is $O(|S_A| \times |\varphi|)$, the total number of checks of the $\forall\exists\exists$ and the $\forall\forall\exists$ conditions is $O(|S_A|^2 \times |\varphi|)$. This is the dominant part which determines the model checking complexity.

Example 7.23. Consider Example 7.1, where the purpose is to verify $\Box p \wedge \Box q$ in the concrete state s_c, abstracted by s_a (see Figure 7.1). This makes s_a the designated state. In this case $\widetilde{R} = \{(s_a, s_{1a}), (s_a, s_{2a})\}$. Thus, we obtain the MC-graph depicted

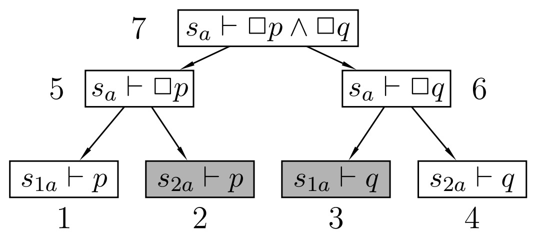


Figure 7.3: A colored model checking graph. Gray vertices are colored ?, while white vertices are colored T.

in Figure 7.3, where $s_a \vdash \Box p \wedge \Box q$ is the initial vertex. Each vertex in the MC-graph comprises a (trivial) MSCC. Figure 7.3 also determines an order on the MSCCs, as indicated by the numbering of the vertices. The vertices $s_{1a} \vdash p$, $s_{2a} \vdash p$, $s_{1a} \vdash q$ and $s_{2a} \vdash q$ are colored as terminal vertices in the sons-coloring phase (in some arbitrary order). Their coloring is depicted in Figure 7.3. For example, $s_{1a} \vdash p$ is colored T since $p \in L_A(s_{1a})$, but $s_{2a} \vdash p$ is colored ? since both $p \notin L_A(s_{2a})$ and $\neg p \notin L_A(s_{2a})$. Once $s_{1a} \vdash p$ is colored T, it is checked if the set $\{s_{1a} \vdash p\}$, which consists of the currently T-colored sons of the vertex $s_a \vdash \Box p$, forms a may hyper-son of $s_a \vdash \Box p$. This is checked by checking if the $\forall\forall\exists$-condition holds for s_a and the set $\{s_{1a}\}$. Since the condition holds, $s_a \vdash \Box p$ is colored T. Similarly, once $s_{2a} \vdash q$ is colored T, it is checked if the set $\{s_{2a} \vdash q\}$ forms a may hyper-son of $s_a \vdash \Box q$ (by checking the $\forall\forall\exists$-condition). Since the condition holds, $s_a \vdash \Box q$ is colored T. Thereafter, $s_a \vdash \Box p \wedge \Box q$ is colored T, and we conclude that the value of $\Box p \wedge \Box q$ in s_a is tt, thus s_c satisfies $\Box p \wedge \Box q$. In fact, the abstract model checking has "discovered" the two may hyper-transitions $(s_a, \{s_{1a}\})$ and $(s_a, \{s_{2a}\})$, which are the ones needed for the verification of the formula in this example.

7.5 Abstraction-Refinement

Our abstract model checking ensures maximal precision. Still, its result might be indefinite if the abstraction is not precise enough. In this case, refinement can be applied by splitting the abstract states, similarly to the refinement of [74] for KMTSs (adapted to the alternation-free μ-calculus and with various optimizations that exploit the use of hyper-transitions).

When refinement is introduced, monotonicity in the precision of the abstract models before and after the refinement is desirable, meaning that formulas that had a definite value before the refinement will not become indefinite after refinement (see Chapter 6). This is guaranteed by the following theorem.

Theorem 7.24 (Monotonicity of HTSs). *Let M_C be a concrete model and let M'_A and M_A be two exact HTSs defined based on abstractions (S'_A, γ') and (S_A, γ) resp., where (S'_A, γ') is a split of (S_A, γ), as defined in Definition 6.10. Then whenever $s'_a \in S'_A$ is a substate of $s_a \in S_A$ then $(M'_A, s'_a) \preceq (M_A, s_a)$.*

Proof. Suppose that $s'_a \in S'_A$ is a substate of $s_a \in S_A$. We show that $(M'_A, s'_a) \preceq (M_A, s_a)$. For this purpose we show that $H \subseteq S'_A \times S_A$ defined by $(s'_a, s_a) \in H$ iff s'_a

is a substate of s_a (see Definition 6.10) is a hyper mixed simulation. Let $(s'_a, s_a) \in H$. We show that the three requirements hold. For requirements 1 and 3, which refer to the labeling and to the must hyper-transitions, the proof is identical to the proof of Theorem 6.11 which refers to the monotonicity of GTSs. This is because these requirements are identical to the requirements of a generalized mixed simulation, and in addition the construction of the labeling function and must hyper-transitions in an exact HTS is the same as in an exact GTS. We now prove that requirement 2 holds as well.

2. Suppose $s_a R^-_A A_a$. Then by the construction, $\forall s_c \in \gamma(s_a)\ \forall s'_c\ [\ s_c R s'_c \Rightarrow \exists s_{a1} \in A_a$ s.t. $s'_c \in \gamma(s_{a1})\ ]$, i.e. $\forall s_c \in \gamma(s_a)\ \forall s'_c\ [\ s_c R s'_c \Rightarrow s'_c \in \gamma(A_a)\ ]$. Since s'_a is a substate of s_a, then $\gamma'(s'_a) \subseteq \gamma(s_a)$, thus in particular $\forall s_c \in \gamma'(s'_a)\ \forall s'_c\ [\ s_c R s'_c \Rightarrow s'_c \in \gamma(A_a)\ ]$. Let $A'_a \subseteq S'_A$ be the set consisting of all the substates of states in A_a. By definition of a split, $\gamma'(A'_a) = \gamma(A_a)$. Therefore the following holds: $\forall s_c \in \gamma'(s'_a)\ \forall s'_c\ [\ s_c R s'_c \Rightarrow s'_c \in \gamma'(A'_a)\ ]$. This implies that $s'_a R^-_{A'} A'_a$. Moreover, $(A'_a, A_a) \in H^{\forall\exists}$ since for every $s'_{a1} \in A'_a$, at least one of its superstates s_{a1} is in A_a (otherwise s'_{a1} would not be included in A'_a), and as such $(s'_{a1}, s_{a1}) \in H$. Thus $\forall s'_{a1} \in A'_a\ \exists s_{a1} \in A_a : (s'_{a1}, s_{a1}) \in H$.

$\square$

Monotonicity implies that refinement of an exact HTS will never take us further from the (definite) result. In particular, we will not "miss" the opportunity to get a definite result only due to excess refinement. Thus, our approach, which is as precise as using the exact HTS w.r.t. the desired property, will ensure the same.

Moreover, if the concrete model M_C is finite, an iterative abstraction-refinement is guaranteed to terminate with a definite answer:

Theorem 7.25. *For finite concrete models, iterating the suggested abstraction-refinement process is guaranteed to terminate with a definite answer.*

7.6 Handling Multiple Initial States

So far we considered the verification problem of a formula in a specific concrete state. We now extend the discussion to the case where the concrete Kripke structure M_C has a *set* of initial states, denoted S^0_C, with the usual meaning, that M_C satisfies φ, denoted $M_C \models \varphi$, if $\forall s_0 \in S^0_C : [\![\varphi]\!]^{M_C}(s_0) = \text{tt}$. Otherwise, M_C falsifies φ, denoted $M_C \not\models \varphi$.

Typically, when the concrete model has a set of initial states, so does the abstract model. For example, in a KMTS or a GTS (see Chapters 2 and 6), in order to ensure a (generalized) mixed simulation from M_C to M_A, the set of abstract initial states S^0_A has to be some set such that

$\forall\exists$(1) : $\forall s_{0c} \in S^0_C\ \exists s_{0a} \in S^0_A$ s.t. $s_{0c} \in \gamma(s_{0a})$, and

$\forall\exists$(2) : $\forall s_{0a} \in S^0_A\ \exists s_{0c} \in S^0_C$ s.t. $s_{0c} \in \gamma(s_{0a})$.

$\forall\exists$(1) is needed to preserve truth, as it ensures that the initial states of the abstract model represent *all* the concrete initial states. On the other hand, $\forall\exists$(2) is needed to preserve falsity, as it ensures that each abstract initial state represents at least one concrete initial state. For example, in Chapter 2, S_A^0 is built such that $s_{0a} \in S_A^0$ iff $\exists s_{0c} \in S_C^0$ s.t. $s_{0c} \in \gamma(s_{0a})$.

This construction is precise if the abstract states represent disjoint sets of concrete states. Yet, similarly to the imprecision introduced by the may transitions when the abstract states are not necessarily disjoint, the same problem occurs w.r.t. the initial states of an abstract model.

In particular, suppose that some concrete initial state s_{0c} is represented by two abstract states: s_a in which φ_1 is true, but φ_2 is indefinite, and s'_a in which φ_2 is true, but φ_1 is indefinite. Then considering s_a as the only initial state will enable verification of φ_1 but not φ_2, and vice versa for s'_a. Yet, no choice of a set of initial states will enable verification of both formulas, even if s_{0c} is the only initial state: including s_a in S_A^0 will prevent verifying φ_2 and including s'_a will prevent verifying φ_2.

This example demonstrates that sometimes different sets of initial abstract states need to be considered for different properties. Therefore, to get a precise abstract model, one needs to allow multiple sets of initial states, with the meaning that any one of them suffices to verify or falsify a property.

Thus, rather than a set of initial states, the class of HTSs is extended by a set of sets of initial states $\mathcal{S}^0 \subseteq 2^{S_A}$, with the meaning that each of the sets in $\mathcal{S}^0$ is a "legal" set of initial states, i.e., it satisfies the $[\forall\exists(1)]$ and $[\forall\exists(2)]$ conditions. In the exact HTS M_A^E, $\mathcal{S}^0$ will consist of *all* the sets that satisfy these conditions.

An extended HTS satisfies φ, denoted $M_A \models \varphi$, if there exists $S_A^0 \in \mathcal{S}^0$ where all the states satisfy φ. It falsifies φ, denoted $M_A \not\models \varphi$, if there exists $S_A^0 \in \mathcal{S}^0$ where at least one state falsifies φ. Otherwise the value of φ in M_A is indefinite, denoted $M_A \overset{?}{\models} \varphi$. Provided that the sets in $\mathcal{S}^0$ fulfill conditions $\forall\exists(1)$ and $\forall\exists(2)$, this ensures preservation of both truth and falsity from an extended HTS to the concrete model it represents.

Here again, instead of checking for each possible set of abstract states if it should be included in $\mathcal{S}^0$ (which requires two $\forall\exists$ checks), and then checking if it enables verification or falsification of φ, one may use a similar technique as was used for the hyper-transitions and choose the candidates more carefully.

The idea is to apply the previous model checking algorithm by setting S^d to $\{s_a \mid \exists s_{0c} \in S_C^0 \text{ s.t. } s_{0c} \in \gamma(s_{0a})\}$. This is the maximal set that fulfills condition $\forall\exists(2)$. Thus, the sets in $\mathcal{S}^0$ in the exact HTS are exactly all the subsets of S^d that fulfill $\forall\exists(1)$ (including S^d itself). When the coloring is over, we do the following.

1. If at least one initial vertex $v \in S^d \times \{\varphi\}$ is colored F, then $M_C \not\models \varphi$. This is because $v = s_a \vdash \varphi$ for some $s_a \in S^d$. Since $S^d \in \mathcal{S}^0$ this implies that $M_A^E \not\models \varphi$, thus $M_C \not\models \varphi$.

2. Otherwise, let ${S^0}^T = \{s_a \in S^d \mid v = s_a \vdash \varphi \text{ is colored } T\}$ be the set of underlying states of the initial vertices that are colored T. If ${S^0}^T$ fulfills the $\forall\exists(1)$ condition, then it is a "legal" set of initial states, in which all of the states satisfy φ, meaning that $M_A^E \models \varphi$, and thus $M_C \models \varphi$.

3. If none of the above holds, then $M_A^E \overset{?}{\models} \varphi$, which means that the abstraction is not precise enough. The correctness of this conclusion results from the fact that had

there been a possible set of initial states in $\mathcal{S}^0$ that falsifies φ, then this set would have contained a state from S^d that falsifies φ, in which case the first item would have applied. Similarly, had there been a possible set of initial states that enables verification of φ, then it would have clearly been a subset of ${S^0}^T$, thus the second item would have applied.

7.7 Concluding Remarks

In this chapter we investigate the precision and model checking complexity of 3-valued abstract models that preserve the full μ-calculus.

In order to evaluate precision of models, we suggest a new definition of precision of 3-valued abstract models, which measures the precision of a model compared to the information retained in the abstract states and concretization function. Namely, the abstract states define the "resolution" through which one can look at the concrete states at every point during the inductive evaluation of a property. An abstract model is precise if it enables to verify or falsify every property that the resolution of the abstract states enables to verify or falsify, resp.

Examining previously suggested abstract models using our new definition reveals that may transitions do not always enable to achieve maximal precision. We therefore suggest a new class of models that use may hyper-transitions to over-approximate the concrete transitions. We propose a construction of a precise abstract model from this class.

Hyper-transitions make the size of the model exponential in the number of abstract states. To avoid this exponential blowup, which already exists in previously suggested models that use must hyper-transitions, we suggest a new abstract model checking algorithm for the alternation free μ-calculus, in which the hyper-transitions are computed by need. As a result, the model checking complexity reduces to $O(|S_A|^2 \times |\varphi|)$, without compromising its precision. We believe that similar techniques can be used to develop precise abstract model checking algorithms for the full μ-calculus, with complexity comparable to model checking of ordinary transition systems.

Finally, we incorporate our abstract model checking into an abstraction-refinement algorithm, where the refinement is monotonic in terms of the precision of the models before and after refinement.

Chapter 8

Compositional Verification and 3-Valued Abstractions Join Forces

8.1 Introduction

Two of the most promising approaches to fighting the state explosion problem are abstraction and compositional verification. In this work we join their forces to obtain a novel fully automatic compositional technique that can determine the truth value of the full μ-calculus w.r.t. a given system.

More specifically, we introduce a new ingredient to compositional model checking, which enhances its modularity: we use a 3-valued model checking game graph as a means to exchange information between the components of a system in the points where their composition is indeed necessary, and ignore the parts which can be handled separately. Thus, our approach avoids the construction of the full composition. We then develop a compositional, incremental, and fully automatic abstraction-refinement framework, which benefits from the modular model checking, and where the refinement is also applied to each component separately. When viewing abstractions as assumptions, our compositional abstraction-refinement has some resemblance to iterative Assume-Guarantee (AG) reasoning. From the AG point of view, our approach can be viewed as a new, automatic, mechanism for assumption generation, which uses the power of abstraction-refinement, and is applicable to the full μ-calculus.

We first present our method for concrete systems, composed of concrete (unabstracted) components. We then extend it to abstract systems, in which one or both of the components have been abstracted (separately). In both cases we avoid the construction of the full system and compose only the parts of the components in which the value of the checked formula remained inconclusive. For simplicity we refer to systems that consist of two components $M_1 \| M_2$. However, our approach can be extended to the composition of n components.

In our setting M_1 and M_2 are Kripke structures that synchronize on the joint labeling of the states. This means that a state of one model is composed with all the states of the other that agree with it on the joint labels. This composition is suitable for modeling synchronous systems with shared variables. In particular, it is suitable for hardware designs that synchronize on their inputs and outputs, since our models can be viewed as Moore machines [42]. The underlying ideas are applicable to other models as well, such as Labeled Transition Systems (LTSs), where components synchronize on their joint transitions and interleave their local transitions (see Section 8.6).

Given a system $M = M_1 \| M_2$, we view each component M_i as an abstract model $M_i\!\uparrow$ of the global system M, in which the values of the local (unshared) variables and the transitions of the other component are unknown. We consider the 3-valued semantics of the μ-calculus, in which the value of a formula in a model is either tt (true), ff (false), or $\perp$ (indefinite). The abstract component $M_i\!\uparrow$ is defined so that whenever a μ-calculus formula φ has a definite value (tt or ff) on $M_i\!\uparrow$, the same value holds also for M. Thus, φ can be checked on either $M_1\!\uparrow$ or $M_2\!\uparrow$ (or both), and if any of them returns a definite result, then this result holds also for M. Only if both checks result in $\perp$, the value of φ in M is unknown.

For the 3-valued abstraction, when the model checking returns $\perp$, the abstract model should be *refined* in order to eliminate the $\perp$ result. For our framework, a refinement could be achieved by composing $M_1\!\uparrow$ and $M_2\!\uparrow$. This, however, is not desired and not necessary. Instead, only the parts of the abstract models for which the model checking result is $\perp$ are identified and composed. The resulting refined model is often significantly smaller than the full system and is guaranteed to return the correct model checking result.

To achieve this goal, our approach uses the game-based 3-valued model checking of the μ-calculus suggested in Chapters 4 and 5. Recall that the vertices of the 3-valued game for model checking (see Chapter 3) are labeled by $s \vdash \psi$, where s is a state of the checked model and ψ is a subformula of the checked formula, such that the value of ψ in s is relevant for determining the model checking result. The model checking algorithms derived from the game determine for each such vertex the truth value of ψ in s based on the 3-valued semantics. We first apply the model checking algorithm to each component separately. If the truth value for s in $M_1\!\uparrow$ is tt (ff), then it is guaranteed that every state in the composed system M, whose first component is s, satisfies (falsifies) ψ. A similar property holds for $M_2\!\uparrow$. Thus, when the model checking returns $\perp$ then only the subgraphs of vertices where $\perp$ results were obtained require further checking and are therefore composed.

The advantage of our approach is that instead of constructing the composition of $M_1\!\uparrow$ and $M_2\!\uparrow$, it focuses on the parts of the components in which their composition is indeed necessary to conclude the truth value of the checked property, due to dependencies between them. It ignores the parts which can be handled separately. Furthermore, if a certain formula only depends on one component, then it can be resolved on this component alone while avoiding the composition altogether. Our technique is orthogonal to the AG approach, and can also be applied when the composed system consists of a component and an assumption on its environment.

To further reduce the size of the checked components, we combine our compositional approach with abstraction. Abstraction not only reduces the state-space of the components, but also allows to handle infinite-state components by abstracting them into finite-state components. Given a system composed of two (or more) components, we first abstract each component separately. However, in order to guarantee preservation of both tt and ff we require that the common alphabet (e.g. common inputs and outputs for hardware designs) will not be abstracted. Only local (unshared) variables can be abstracted. While this limits the amount of reduction that can be achieved by the abstraction on a single component, it enables additional reduction due to the compositional reasoning.

We propose an automatic construction of the initial abstraction for each component separately. We then proceed as before: we run a 3-valued model checking on each of the components. If both return $\perp$, then we identify and compose the parts where indefinite results were obtained, and apply 3-valued model checking to the composed model. While in the concrete case this step always terminates with a definite result, here we may obtain an indefinite result due to abstraction. In such a case, we follow Chapters 4 and 5 in finding the *cause* for the indefinite result on the composed model. However, the refinement itself is applied on each of the components separately. Moreover, we adopt the incremental approach of Chapters 4 and 5 and refine only the indefinite part of each component. Thus, results from previous iterations are re-used. The result is a compositional abstraction-refinement framework.

An abstraction of a component M_i (which comprises the environment of the other component) can be viewed as providing an assumption on M_i. From this point of view, when applying abstraction-refinement on one or both of the components, the result is an automatic mechanism for assumption generation, which is either symmetric (refers to both components) or asymmetric (abstracts only one component). In each iteration, more information about the component is revealed, by need – based on the cause for the indefinite result. This resembles iterative AG reasoning. The use of conservative abstractions guarantees that the assumption describes the component correctly (by construction). Thus unlike typical AG reasoning, this need not be verified. Moreover, our approach benefits from the modular model checking described above.

8.1.1 Related Work

Many of the works on compositional model checking are based on the Assume-Guarantee (AG) paradigm [46, 68]. Recently, [23] followed by [5, 15, 34], considered automatic assumption generation for AG reasoning. They use *learning* algorithms for finite automata in order to automatically produce suitable assumptions for an AG rule. A similar approach is taken in [9], where the AG rule used is symmetric. Assumption generation in a more general setting (not necessarily for AG reasoning) is addressed in [36]. The work of [2] on interface synthesis for application programs can also be seen as assumption generation. These works are all restricted to *universal safety* properties (either in a linear time or a branching time setting). More recently, [34] extended the learning-based approach to liveness properties as well (in a linear time setting), by proposing a learning algorithm for the full class of ω-regular languages. The learning algorithms used in these works also perform some kind of an abstraction-refinement. However, these algorithms are not specifically tailored for verification. In particular, they do not always maintain a conservative abstraction of the environment. As such, the assumption sometimes needs to be weakened and sometimes needs to be strengthened. In contrast, our approach is based on techniques taken from the 3-valued game-based model checking for abstract models described in the previous chapters. In our case an assumption (abstraction) should never be weakened. Moreover, we increase the modularity of the model checking step by using the game-based approach, which also enables an incremental analysis.

Most importantly, our approach is applicable to the *full* (branching-time) μ-calculus.

The game-based model checking enables us to identify the places where the value of a subformula in a component's state is the same for *all* environments. We exploit this information to reduce the model checking instance of the entire system. Other authors have also used similar information for reductions. In [1] the authors merge component's states that share the same value for a given CTL formula in all environments, thus minimizing the component. In [3] the authors use reachability and controllability information about the concrete components (gathered via game-theoretic techniques) in order to construct abstract components for *invariance* properties. The composition of the abstract components is then computed and model checked. We, on the other hand, do not try to minimize each component. Instead, the game graph enables us to prune parts of each component's model checking instance whose effect was already taken into consideration. As a result, we reduce the state space exploration of the entire system. This is applicable even if no states of the individual components can be merged.

[30] uses controllability information to speed up falsification of invariance properties. They identify unpreventable violations of the property based on each component separately, which enables to prune the state space exploration of the compound system before a violation is actually encountered. The authors state that their method can be extended to arbitrary LTL properties. However, they only use controllability information w.r.t. the entire formula. Our approach enables to gather information about subformulas as well, and thus can result in more substantial reductions. In addition, our approach is aimed at both verification and falsification (with a 3-valued semantics) and is applicable to a full branching time logic.

[59] also uses 3-valued model checking for modular verification. They consider feature-oriented modules, where the composition is via interfaces and has a more sequential nature. As a result, they only refer to unknown propositions and not to uncertainty in the transitions. A substantial part of their work is devoted to determining what information needs to be included in a feature's interface to support compositional reasoning. In our case, we use the game graph for sharing such auxiliary information about the individual components.

In [4] the authors suggest to use game structures to reason about composition of components. [29, 7] suggest abstraction-refinement frameworks for such models, w.r.t. alternating time temporal logics, which enable to describe properties of the interaction between components. We are interested in properties of the compound system, thus the focus in these works is different. In addition, they abstract each component separately and then model check the entire system. The model checking step is not modular.

[14] develops a compositional counterexample-guided abstraction refinement for a universal temporal logic (which extends ACTL). In their approach, the abstraction and the refinement steps are performed on each component separately, but the model checking step is done on the entire (abstract) system. In our approach, the model checking step is also compositional, and the properties considered are not limited to a universal logic.

8.2 Preliminaries

In this chapter we consider KMTSs as abstract models, rather than other formalisms. This is because KMTSs come up very naturally when viewing a component of a system as an abstract model of the entire system (see Section 8.4).

The compositional approach developed in this chapter utilizes the game-based 3-valued model checking algorithms of Chapters 4 and 5. In this section we briefly remind the reader of these algorithms. The description of the algorithms is customized in order to make the presentation of the compositional approach simpler. It slightly differs from the original presentation.

We consider KMTSs (in particular Kripke structures) with a single initial state. As such, we describe the 3-valued model checking game and the derived model checking algorithms for the initial state of the KMTS. In the following, let $M = (S, s^0, R^+, R^-, L)$ be a KMTS and $\varphi \in \mathcal{L}_\mu$.

8.2.1 Game Graph

Consider the 3-valued model checking game $\Gamma_M(s^0, \varphi)$ of $\varphi \in \mathcal{L}_\mu$ over the initial state s^0 of the KMTS M (see Section 3.2). The game induces a *game graph*, denoted $G = (V, v^0, E^+, E^-)$, where $V \subseteq S \times Sub(\varphi)$ is the set of vertices, $v^0 = s^0 \vdash \varphi$ is the initial vertex, and $E^+ \subseteq E^- \subseteq V \times V$ are sets of must and may edges. The vertices of the game graph consist of the reachable configurations in the game, classified as $\wedge$, $\vee$, $\Box$, $\Diamond$, literal, or deterministic vertices, based on the classification of the configurations. The edges are defined by the moves of the game (see Figure 3.1) and are classified as must vs. may edges accordingly: The edges that are based on a may transition of M which is not a must transition are genuine may edges in $E^- \setminus E^+$ and all the rest are both must and may edges, i.e., belong to both E^+ and E^-. The game graph G is isomorphic to the arena of the 3-valued parity game associated with $\Gamma_M(s^0, \varphi)$, except that no distinction is made between V_0, V_1, and V_{tie}. Moreover, for convenience we refer to the vertex $s^0 \vdash \varphi$ as the initial vertex of the game graph.

Figure 8.2(b) presents examples of the game graphs of the 3-valued model checking games for $\varphi = \Box(\neg i \vee \Diamond o)$ and the models from Figure 8.2(a), where all transitions are considered may transitions.

8.2.2 Coloring Algorithm

The model checking algorithms described in Chapters 4 and 5 can be viewed as coloring algorithms[1] that label (color) each vertex $v = s \vdash \psi$ in the game graph of the 3-valued model checking game by T, F, ? depending on the player that has a winning strategy in the game, or equivalently depending on the truth value of ψ in the state s in M (based on the 3-valued semantics). The result of the coloring is a *3-valued coloring function* $\chi : V \to \{T, F, ?\}$.

In both Chapter 4 and Chapter 5 the coloring is performed by solving the 3-valued parity game that corresponds to the model checking game, where each color stands for

[1]We refer to the model checking algorithms of Chapters 4 and 5 as coloring algorithms although they were not originally described in these terms.

a possible result (winner) in the game. In fact, any other algorithm for solving 3-valued parity games can be used as well.

However, for simplicity, we refer to these algorithms as coloring algorithms which are applied on the game graph G induced by the 3-valued model checking game. This is justified by the observation that in the context of the 3-valued model checking game, the game graph carries all the information regarding the corresponding parity game. Namely, the game graph is isomorphic to the arena of the parity game, and the subformulas of the vertices uniquely determine both the classification of the vertices to V_0, V_1 and V_{tie} and the priorities of the vertices.

The coloring is correct if the color that it assigns to each vertex corresponds to the player that has a winning strategy from that vertex. In the context of 3-valued model checking games, this also relates to the semantics. We therefore use the following definition. For a (possibly not closed) formula ψ, ψ^* denotes the result of replacing every free occurrence of a variable $Z \in \mathcal{V}$ in ψ by $fp(Z)$, until no free variable remains. ψ^* is always a closed formula. Note that if ψ is closed, then $\psi^* = \psi$.

Definition 8.1. *Let G be the game graph of the 3-valued model checking game for a KMTS M with initial state s^0 and $\varphi \in \mathcal{L}_\mu$. A (possibly partial) coloring function $\chi : V \to \{T, F, ?\}$ for G (or its subgraph) is* correct *if for every $s \vdash \psi \in V$, whenever $\chi(s \vdash \psi)$ is defined, then it matches the semantics, i.e.:*

1. $[\![\psi^*]\!]_3^M(s) = \mathrm{tt}$ *iff* $\chi(s \vdash \psi) = T$.
2. $[\![\psi^*]\!]_3^M(s) = \mathrm{ff}$ *iff* $\chi(s \vdash \psi) = F$.
3. $[\![\psi^*]\!]_3^M(s) = \perp$ *iff* $\chi(s \vdash \psi) = ?$.

Theorem 3.3 relates the truth value of φ in s^0 to the player that has a winning strategy in the model checking game $\Gamma_M(s^0, \varphi)$. Therefore, along with the correctness of the algorithms for solving 3-valued parity games, it ensures the correctness of the color of the initial vertex v^0. In this chapter we will also be interested in the internal vertices of the game graph. We therefore generalize the correspondence between the 3-valued semantics and the coloring to *all* the vertices in the game graph.

Theorem 8.2. *Let χ_F be the (total) coloring function returned by the coloring algorithm of Chapter 4 or Chapter 5 for the game graph G. Then χ_F is correct.*

Proof. It suffices to prove that the correspondence between the truth value of ψ^* in s and the player that has a winning strategy starting from the vertex $s \vdash \psi$ in the model checking game $\Gamma_M(s^0, \varphi)$ is maintained when talking about internal vertices $s \vdash \psi \in V$, and not just when talking about the initial vertex (as described in Theorem 3.3). The correctness of the algorithms for solving 3-valued parity games will then imply the correspondence to the color.

Note that the player that has a winning strategy in the model checking game $\Gamma_M(s^0, \varphi)$ starting from the vertex $s \vdash \psi$ is the same as the player that has a winning strategy in the game $\Gamma_M(s, \psi)$. Therefore, we equivalently relate the truth value of ψ^* in s to the player that has a winning strategy in $\Gamma_M(s, \psi)$.

For closed subformulas ψ of φ the claim is immediately implied by Theorem 3.3 (since $\psi^* = \psi$ in this case). We now consider subformulas which are not closed. We first show that replacing a free occurrence of a variable $Z \in \mathcal{V}$ in ψ by $fp(Z)$ does not change the player that has a winning strategy in the game. Namely, the winner in the game $\Gamma_M(s, \psi)$ is the same as the winner in the game $\Gamma_M(s, \psi[Z := fp(Z)])$. This is because the only difference between the games is an additional deterministic move in the latter one from vertices of the form $s' \vdash fp(Z)$ to $s' \vdash Z$. Clearly, deterministic moves do not change the winner of the game.

Iterating this argument implies that the player that has a winning strategy in the game $\Gamma_M(s, \psi)$ is the same as the player that has a winning strategy in the game $\Gamma_M(s, \psi^*)$. By Theorem 3.3, the latter corresponds to the truth value of ψ^* in s, which concludes the proof. $\square$

The result of the coloring of the game graphs for $\varphi = \Box(\neg i \vee \Diamond o)$ and the models from Figure 8.2(a) is demonstrated in Figure 8.2(b).

8.3 Partial Coloring and Subgraphs

In the following sections we use the game-based model checking in order to identify and focus on the places where the dependencies between components of the system affect the model checking result. In this section we set the basis for this, by investigating properties of the game graph and the coloring algorithms.

Specifically, the coloring algorithms of Chapters 4 and 5 have the important property that they can be applied on a partially colored graph, in which case they extend the given coloring to the rest of the graph in a correct way. Moreover, the coloring can also be applied on a partially colored *subgraph*, and under certain assumptions it will yield a correct coloring of the subgraph. We now formalize this property.

Recall that in the 3-valued parity game that corresponds to the 3-valued model checking game, Player 0 takes the role of the verifier and Player 1 takes the role of the falsifier. In the final coloring of the game graph, vertices where Player 0 wins are colored T, vertices where Player 1 wins are colored F and the rest are colored ?.

A partially colored subgraph G' of the game graph G of a 3-valued model checking game also induces a 3-valued parity game. The *induced game* differs from the original one in the omission of the vertices (and edges) outside G'. In addition, the vertices which are colored T by the initial coloring function become terminal vertices in V_1 (i.e., winning for Player 0), the vertices colored F become terminal vertices in V_0 (i.e., winning for Player 1), and the vertices colored ? become terminal vertices in V_{tie} (i.e. not winning for both players). This ensures that the winner of the already colored vertices in the induced game corresponds to their color. The priorities remain the same.

Due to their nature, as algorithms for solving a 3-valued parity game, the coloring algorithms can be applied on the induced game (which is also a 3-valued parity game). Similarly to before, instead of referring to the algorithms as applied on the induced game, we simply refer to them as applied on G' with an initial coloring function since these carry all the information regarding the induced game. To formalize the conditions that ensure the correctness of the coloring we need the following definitions.

Definition 8.3. *Let G be the game graph of a 3-valued model checking game and χ_F its final coloring function. For a non-terminal vertex v in G we define its* witnessing sons *as follows, depending on its type:*

$\wedge, \Box$: *the witnessing sons are those colored F or ? by χ_F.*

$\vee, \Diamond$: *the witnessing sons are those colored T or ? by χ_F.*

deterministic: *the witnessing son is the only son.*

The sons are witnessing in the sense that they suffice to determine the color (winner) of the vertex, thus removing the rest of the vertex's sons from the graph does not damage the result of the coloring. More specifically, when considering the game induced by the graph after the remaining vertex's sons are removed, the winner (and thus the color) remains unchanged. This is because the sons which are not witnessing will never be used in a winning (or even non-losing) strategy: In a $\wedge$-vertex or a $\Box$-vertex v, where Player 1 moves, he will never choose a son colored T (where Player 0 has a winning strategy) as part of a winning or non-losing strategy. In particular, if v has no witnessing sons, meaning all its sons are colored T (i.e., won by Player 0), then Player 1 simply has no winning (nor non-losing) strategy from v, i.e., the winner in v is Player 0 and it should be colored T. This is also the winner (color) in the induced game when keeping only the witnessing sons (i.e., when no sons remain and the vertex becomes a terminal vertex in V_1). Similarly for a F-colored son of a $\vee$-vertex or a $\Diamond$-vertex, where Player 0 moves.

Definition 8.4. *A subgraph G' of a game graph G of a 3-valued model checking game is* closed *if every vertex in G' is either a terminal vertex, or* all *its witnessing sons (and corresponding edges) from G are also in G'.*

Theorem 8.5. *Consider a closed subgraph G' of a game graph G of a 3-valued model checking game with a partial coloring function χ which is correct and defined over (at least) all the terminal vertices in G'. Then applying the coloring algorithm of Chapter 4 or Chapter 5 on G' with χ as an initial coloring results in a correct coloring of G'.*

Proof. It suffices to show that the winner of each vertex in G' is the same both in the induced game and in the full game. This ensures the correctness of the coloring which is performed by solving the 3-valued parity game induced by G'.

For the terminal vertices of G' this is clear, since the terminal vertices are classified such that their winner corresponds to their initial color and the initial coloring of the terminal vertices is correct (see Definition 8.1). Thus, a terminal vertex in V_0 is a vertex that was colored F in the full graph, meaning that Player 1 wins from it in both games. Similarly for the terminal vertices in V_1.

As for the non-terminal vertices in G', we first recall that $\vee$ and $\Diamond$ vertices are controlled by Player 0, whereas $\wedge$ and $\Box$ vertices are controlled by Player 1. For $\vee$ and $\Diamond$ vertices, controlled by Player 0, the non-witnessing sons are colored F, which means they are winning for Player 1 in the full game, thus Player 0 will not use them in a winning strategy. Dually, For $\wedge$ and $\Box$ vertices, controlled by Player 1, the non-witnessing sons are colored T, which means they are winning for Player 0, thus Player 1 will not use them in a winning strategy.

To show that the winner of each non-terminal vertex in G' is the same in both games, we show that each winning strategy in the full game translates to a winning strategy of the same player in the induced game, and vice versa. Consider such a vertex v:

Suppose that Player σ has a winning strategy in the full game starting at v. Then, the same strategy is a winning strategy in the induced game, with the exception that in the "new" terminal vertices of G' the strategy is "pruned", and the winner remains the same due to the above claim regarding the terminal vertices of G'. Note that the winning strategy in the full game never uses sons which are not witnessing (since as explained above, in the vertices controlled by Player σ the non-witnessing sons are not winning for σ). Therefore the strategy is well defined in the induced game as well.

For the opposite direction, suppose that Player σ has a winning strategy in the induced game starting at v. Then the same strategy is a winning strategy in the full game, except that starting from the "new" terminal vertices of G' the "original" strategy is used – again, due to the above claim the winner there remains the same. Moreover, starting from the non-witnessing sons of vertices controlled by Player $\overline{\sigma}$ (which are not present in the induced game), the original winning strategy of σ is used (as explained above such non-witnessing sons are winning for σ in the full game), and the winner remains the same. □

In fact, for the coloring of the subgraph to be correct, not *all* the witnessing sons are needed, as long as there is enough information to explain the correct coloring of each uncolored vertex. However, we will see that in our case we will need all of them, as we will deduce from the game graph of one component to the game graph of the full system, where some of the vertices will be removed and for some an indefinite color (?) will change into T or F. This means that some of the witnessing sons will not remain witnessing sons in the game graph of the full system. Thus, we will not be able to know a-priori which of them is the "right" choice to include in a way that will also provide the necessary information for a correct coloring in the game graph of the full system.

Another notion that we will need later is the following.

Definition 8.6 (**?**-Subgraph)**.** *Let G be a colored graph whose initial vertex is colored ?. The* ?-subgraph *is the least subgraph $G_?$ of G that obeys the following:*

- *the initial vertex is in $G_?$ (and is the initial vertex of $G_?$).*
- *For each vertex in $G_?$ which is colored ? in G all its witnessing sons (and corresponding edges) in G are included in $G_?$.*

$G_?$ is accompanied with a partial coloring function χ_I which is defined over the terminal vertices in $G_?$, and colors them as the coloring function χ_F of G.

The ?-subgraph $G_?$ and its initial coloring meet the conditions of Theorem 8.5. Intuitively, this means that $G_?$ contains *all* the information regarding the indefinite result. Figure 8.2(b) provides examples of ?-subgraphs.

8.4 Compositional Model Checking

In compositional model checking the goal is to verify a formula φ on a compound system $M_1 \| M_2$. In our setting M_1 and M_2 are Kripke structures that synchronize on the joint labeling of the states. Since a Kripke structure is a special case of a KMTS where $R = R^+ = R^-$, we define the composition for the more general case of KMTSs. In the following we denote by Lit_1 and Lit_2 the sets of literals over AP_1 and AP_2, resp.

Definition 8.7. *Two KMTSs $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ are* composable *if their initial states agree on their joint labeling, i.e. $L_1(s_1^0) \cap Lit_2 = L_2(s_2^0) \cap Lit_1$.*

Definition 8.8. *Let $M_1 = (AP_1, S_1, s_1^0, R_1^+, R_1^-, L_1)$ and $M_2 = (AP_2, S_2, s_2^0, R_2^+, R_2^-, L_2)$ be two composable KMTSs. We define their composition, denoted $M_1 \| M_2$, to be the KMTS $(AP, S, s^0, R^+, R^-, L)$, where*

- $AP = AP_1 \cup AP_2$
- $S = \{(s_1, s_2) \in S_1 \times S_2 \mid L_1(s_1) \cap Lit_2 = L_2(s_2) \cap Lit_1\}$
- $s^0 = (s_1^0, s_2^0)$
- $R^+ = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^+ \text{ and } (s_2, t_2) \in R_2^+\}$
- $R^- = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1^- \text{ and } (s_2, t_2) \in R_2^-\}$
- $L((s_1, s_2)) = L(s_1) \cup L(s_2)$

In particular, if M_1 and M_2 are Kripke structures with transition relations R_1 and R_2 resp., then $M_1 \| M_2$ is a Kripke structure with $R = \{((s_1, s_2), (t_1, t_2)) \in S \times S \mid (s_1, t_1) \in R_1 \text{ and } (s_2, t_2) \in R_2\}$.

From now on we fix AP to be $AP_1 \cup AP_2$. For $i \in \{1, 2\}$ we use $\bar{i}$ to denote the remaining index in $\{1, 2\} \setminus \{i\}$.

We use the mechanism produced for abstractions of full branching time logics for the purpose of compositional verification. The basic idea is to view each Kripke structure M_i as a partial model that abstracts the complete system $M_1 \| M_2$.

Definition 8.9. *Let $M_i = (AP_i, S_i, s_i^0, R_i, L_i)$ be a Kripke structure. We lift M_i into a KMTS $M_i\!\uparrow = (AP, S_i, s_i^0, R_i^+\!\uparrow, R_i^-\!\uparrow, L_i\!\uparrow)$ over AP where $R_i^+\!\uparrow = \emptyset$, $R_i^-\!\uparrow = R_i$ and $L_i\!\uparrow(s) = L_i(s)$.*

That is, we view M_i as a KMTS $M_i\!\uparrow$ over AP (rather than AP_i). This immediately makes the value of each literal over $AP \setminus AP_i$ in each state of $M_i\!\uparrow$ indefinite (as neither p nor $\neg p$ are in $L_i(s)$) – indeed, it depends on $M_{\bar{i}}$. In addition, each transition of M_i is considered a may transition (since in the composition it might be removed if a matching transition does not exist in $M_{\bar{i}}$, but transitions can never be added).

Theorem 8.10. *For each $i \in \{1, 2\}$, $M_1 \| M_2 \preceq M_i\!\uparrow$. The mixed simulation relation $H \subseteq S \times S_i$ is given by $\{((s_1, s_2), s_i) \mid (s_1, s_2) \in S\}$.*

Proof. For a state $s = (s_1, s_2)$ of $M_1 \| M_2$ and $i \in \{1, 2\}$ let $\pi_i(s)$ denote the projection of the pair on its ith component. The mixed simulation relation $H \subseteq S \times S_i$ is given by $\{(s, s_i) \mid \pi_i(s) = s_i\}$. That is, a state s_i of $M_i\!\uparrow$ is related by a mixed simulation relation to all the states of $M_1 \| M_2$ where the corresponding state in the pair is s_i. Clearly, the initial states are in H since by definition of the composition, the initial state of $M_1 \| M_2$ consists of the initial states of both components. Let $(s, s_i) \in H$. Then:

- $L(s) = L_1(\pi_1(s)) \cup L_2(\pi_2(s))$ (by the definition of composition), and the latter is clearly a superset of $L_i(\pi_i(s)) = L_i(s_i)$.

- The requirement for must transitions is met vacuously as there are no must transitions in $M_i\!\uparrow$.

- Let $(s, s') \in R$. Then by definition of the composition, in particular for i, there exists a transition $(\pi_i(s), \pi_i(s')) \in R_i$, meaning that $(\pi_i(s), \pi_i(s')) \in R_i^-\!\uparrow$ (by the definition of $M_i\!\uparrow$). In addition, by the definition of H, $(s', \pi_i(s')) \in H$.

□

Since each $M_i\!\uparrow$ abstracts $M_1 \| M_2$, we are able to first consider each component separately: Theorem 2.5 ensures that if φ has a definite value (tt or ff) in the KMTS $M_i\!\uparrow$ under the 3-valued semantics, then the same value holds in $M_1 \| M_2$ as well. In particular, the values in $M_1\!\uparrow$ and $M_2\!\uparrow$ cannot be contradictory, and it suffices that one of them is definite in order to determine the value in $M_1 \| M_2$.

The more typical case is that the value of φ on both $M_1\!\uparrow$ and $M_2\!\uparrow$ is indefinite. This reflects the fact that φ depends on both components and their synchronization. Typically, an indefinite result requires some refinement of the abstract model. In our case refinement means considering the composition with the other component. Still, in this case as well, having considered each component separately can guide us into focusing on the places where we indeed need to consider the composition of the components.

The game-based approach to model checking provides a convenient way for presenting this information. If the KMTS $M_i\!\uparrow$ is model checked using the 3-valued game-based algorithm of Chapter 4 or Chapter 5, then the result is a colored game graph, in which T and F represent definite results (i.e. truth values that hold no matter what the environment is), but the ? color needs to be resolved by considering the composition. This is where the ?-subgraph (see Definition 8.6) becomes handy, as it points out the places where this is really needed.

The ?-subgraph for each component is computed top-down, starting from the initial vertex. As long as a vertex colored ? is encountered, the search continues in a BFS manner by including the witnessing sons. Definite vertices which are included in the subgraph become terminal vertices, and their coloring defines the initial coloring function.

The ?-subgraphs of the two colored graphs present all the indefinite information that results from the dependencies between the components. Thus, to resolve the indefinite result, we compose the ?-subgraphs.

Definition 8.11 (Product Graph). *Let $G_{?1}$ and $G_{?2}$ be two ?-subgraphs as above with initial vertices $s_1^0 \vdash \varphi$ and $s_2^0 \vdash \varphi$ resp. We define their product to be the least graph $G_\| = (V_\|, v_\|^0, E_\|^+, E_\|^-)$ such that:*

- *$v_\|^0 = (s_1^0, s_2^0) \vdash \varphi$ is the initial vertex in $V_\|$.*

- *If $(s_1, s_2) \vdash \psi \in V_\|$ and $(s_1 \vdash \psi, s_1' \vdash \psi') \in E_1^-$ and $(s_2 \vdash \psi, s_2' \vdash \psi') \in E_2^-$ and $L_1(s_1') \cap Lit_2 = L_2(s_2') \cap Lit_1$ (i.e. (s_1', s_2') is a state of $M_1 \| M_2$), then: $(s_1', s_2') \vdash \psi' \in V_\|$ and $((s_1, s_2) \vdash \psi, (s_1', s_2') \vdash \psi')$ is in $E_\|^+$ and $E_\|^-$.*

Note that all the edges in $G_{\|}$ are must edges, whereas in the ?-subgraphs we had may edges (the transitions of each component were treated as may transitions in the lifted version). This is because the product graph already refers to the complete system $M_1 \| M_2$, where all transitions are concrete transitions (modeled as must transitions).

The product graph is constructed by a top-down traversal of the subgraphs, where, starting from the initial vertices, vertices that share the same formulas and whose states agree on the joint labeling are composed (recall that s_1^0 and s_2^0 agree on their joint labeling). Whenever two non-terminal vertices are composed, the outgoing edges are computed as the product of their outgoing edges, limited to legal vertices (w.r.t. the restriction to states that agree on their labeling). In particular, this means that if a vertex in one subgraph has no matching vertex in the other, then it will be omitted from the product graph. In addition, when a terminal vertex of one subgraph is composed with a non-terminal vertex of the other, the resulting vertex is a terminal vertex in $G_{\|}$.

We accompany $G_{\|}$ with an initial coloring function for its terminal vertices based on the initial coloring functions of the two subgraphs. We use the following observation:

Proposition 8.12. *Let $v = (s_1, s_2) \vdash \psi$ be a terminal vertex in $G_{\|}$. Then one of the following holds. Either (a) at least one of $s_1 \vdash \psi$ and $s_2 \vdash \psi$ is a terminal vertex in its subgraph, in which case at least one of them is colored by a definite color by the initial coloring of its subgraph, and contradictory definite colors are impossible. We denote this color by $col(v)$; Or (b) both $s_1 \vdash \psi$ and $s_2 \vdash \psi$ are non-terminal vertices but no outgoing edges were left in their composition.*

Proof. Clearly, if $s_1 \vdash \psi$ and $s_2 \vdash \psi$ are both non-terminal vertices, then for $v = (s_1, s_2) \vdash \psi$ to be a terminal vertex in $G_{\|}$, it has to be the case that no outgoing transitions were left in the composition of $s_1 \vdash \psi$ and $s_2 \vdash \psi$. This refers to case (b).

As for case (a), if at least one of $s_1 \vdash \psi$ and $s_2 \vdash \psi$ is a terminal vertex in its ?-subgraph, then we show that at least one of them is colored by a definite color: First, by the construction of a ?-subgraph, an indefinite color for a terminal vertex is only possible when its subformula is a literal. This is because if a vertex with any other formula is colored ?, then it has at least one witnessing son which will be included in the ?-subgraph, making the vertex non-terminal. For example, a $\Box$-vertex which is colored ? has at least one son which is colored $\neq T$, and is thus a witnessing son. A literal only has an indefinite value if it refers to a local atomic proposition of the other component, in which case the vertex in the other component has a definite color.

In addition, contradictory definite colors cannot exist due to the correctness of the coloring w.r.t. the 3-valued semantics: if $s_1 \vdash \psi$ is colored T, then the value of ψ in s_1 is tt, and by the mixed simulation relation, its value in (s_1, s_2'), for every s_2' that is composable with s_1, is also tt. By the same arguments, if $s_2 \vdash \psi$ is colored F, then the value of ψ in (s_1', s_2), for every s_1' that is composable with s_2, is also ff. This leads to contradiction since s_1 and s_2 are composable. $\Box$

Definition 8.13. *We define the initial coloring function χ_I of $G_{\|}$ as follows. Let v be a terminal vertex in $V_{\|}$. If it fulfills case (a) of Proposition 8.12, then $\chi_I(v) = col(v)$. If it fulfills case (b), then $\chi_I(v) = T$ if v is a $\wedge$-vertex or a $\Box$-vertex, and $\chi_I(v) = F$ if v is a $\vee$-vertex or a $\Diamond$-vertex. χ_I is undefined for the rest of the vertices.*

In particular, if a terminal vertex in $G_{\|}$ results from a terminal vertex which is colored by ? in one subgraph and a terminal vertex which is colored by some definite color in the other (case (a)), then the definite color takes over.

Note that a terminal vertex that fulfills case (b) cannot be deterministic, thus all the possible cases are covered by the definition of the initial coloring function of $G_{\|}$.

Note further that the initial coloring function of the product graph colors all the terminal vertices by definite colors. Along with the property that all the edges in the product graph are must edges, this reflects the fact that the composition resolves all the indefinite information that existed in each component when it was considered separately. Therefore, when applying (one of) the coloring algorithms to the product graph, all the vertices are colored by definite colors (in fact, a 2-valued coloring can be applied).

Theorem 8.14. *The resulting product graph $G_{\|}$ is a closed subgraph of the game graph over $M_1\|M_2$. In addition, the initial coloring function is correct w.r.t. $M_1\|M_2$ and defined over all the terminal vertices in the subgraph.*

Proof. We first show that $G_{\|}$ is a closed subgraph of the game graph that corresponds to the composition $M_1\|M_2$. It is easy to see that it is a subgraph, since the structure in terms of the subformulas and edges is maintained. We now show that the subgraph is closed. Assume to the contrary that it contains a non-terminal vertex v whose witnessing sons are not all included. Let v' be such a witnessing son. This means that both of the vertices that correspond to v in the ?-subgraphs of the two components are colored by indefinite colors (otherwise, if at least one of them was colored by a definite color, it would have been a terminal vertex and so would v). Moreover, at least one of the vertices that correspond to the witnessing son v' is not included in its subgraph (otherwise v' would have been included in the product graph). Denote this vertex by $\widetilde{v}'$. By the relation of the colors to the 3-valued semantics and by the preservation guarantee of the 3-valued semantics, the color of $\widetilde{v}'$ in its game graph is either the same as the color of v' in the game graph for the composition $M_1\|M_2$, or indefinite. However, in both cases this makes it a witnessing son in its game graph, which means it should have been included in the ?-subgraph. To see why $\widetilde{v}'$ is a witnessing son of $\widetilde{v}$ (the vertex that corresponds to v) in its game graph, first note that if $\widetilde{v}'$ is colored ?, then this is immediate. If $\widetilde{v}'$ is colored by a definite color in its game graph, then, as explained above, its color is the same as the color of v' in the full game graph of $M_1\|M_2$. Therefore, since v' is a witnessing son of v, and since the types of v and $\widetilde{v}$ (resp. v' and $\widetilde{v}'$) are the same, it holds that $\widetilde{v}'$ is also a witnessing son of $\widetilde{v}$.

We show that the initial coloring function is correct by a case analysis. For terminal vertices that are colored based on case (a) this directly results from the preservation guarantee of the 3-valued semantics (Theorem 2.5). As for terminal vertices that are colored based on case (b), consider the case of a vertex v with a $\wedge$ or a $\Box$ formula (the other case is dual). If v is also a terminal vertex in the full game graph of $M_1\|M_2$, then it must consist of a $\Box$ formula and of a state of $M_1\|M_2$ that has no successors, which means it satisfies the $\Box$ formula and the T-color of v is correct. Otherwise, since all the sons of v were removed during the composition, this means that every one of them corresponds to a vertex that does not exist in the ?-subgraph of at least one component, which means it was colored T in the game graph of the component. Again, the preservation theorem

Step 1 Model check each $M_i\!\uparrow$ separately (for $i \in \{1,2\}$):

1. Construct the game graph G_i for φ and $M_i\!\uparrow$.
2. Apply the 3-valued coloring on G_i. Let χ_i be the resulting coloring function.

If $\chi_1(v_1^0)$ or $\chi_2(v_2^0)$ is definite, return the corresponding model checking result for $M_1 \| M_2$.

Step 2 Consider the composition $M_1 \| M_2$:

1. Construct the ?-subgraphs for G_1 and G_2.
2. Construct the product graph $G_\|$ of the ?-subgraphs.
3. Apply the 3-valued coloring on $G_\|$ (with the initial coloring function).

Return the model checking result corresponding to $\chi_\|(v_\|^0)$.

Figure 8.1: Compositional model checking algorithm.

ensures that the color of all the sons in the game graph of $M_1 \| M_2$ is also T, and thus T is the correct color of v. □

By Theorem 8.5, this means that coloring $G_\|$ results in a correct result w.r.t. the model checking of φ in $M_1 \| M_2$. Thus, to model check φ on $M_1 \| M_2$ it remains to color $G_\|$. Note that the full graph for $M_1 \| M_2$ is not constructed. The resulting compositional model checking algorithm appears in Figure 8.1.

Example 8.15. Consider the two components depicted in Figure 8.2(a). The atomic proposition o (short for *output*) is local to the first component M_1, i (*input*) is local to the second component M_2, and r (*receive*) is the only joint atomic proposition that the components M_1 and M_2 synchronize on. Suppose we wish to verify in the compound system $M_1 \| M_2$ the property $\Box(\neg i \vee \Diamond o)$, which states that in all the successor states of the initial state, an *input* signal implies that there is a successor state where the *output* signal holds. Figure 8.2(b) depicts the colored game graph of each (lifted) component, and highlights the ?-subgraph of each of them. The product graph and its coloring is depicted in Figure 8.2(c), as an "intersection" of the two subgraphs. All the edges in the product graph are must edges. All vertices, and in particular the initial vertex, are colored T, thus the property is verified. One can see that most of the efforts were done on each component separately, and the product graph only considers a small part of the compound system.

8.5 Adding Abstraction

In Section 8.4 we considered concrete components. The indefinite results on each component resulted only from their interaction, and were resolved by composing the indefinite

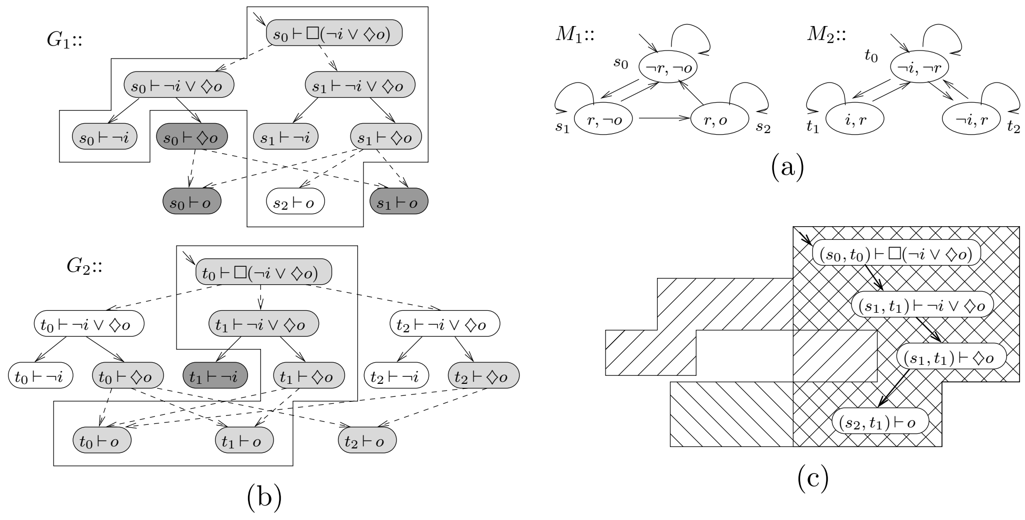


Figure 8.2: (a) Components, (b) their game graphs and their ?-subgraphs (enclosed by a line), and (c) the product graph. Dashed edges denote may edges which are not must edges. The colors reflect the coloring function: white stands for T, dark gray stands for F and light gray stands for ?.

parts. We now combine this idea with existing abstraction-refinement techniques that abstract one or both of the components (separately), and refine them gradually when necessary. This will enable us to combine the benefits of the compositional approach with the power of iterative abstraction-refinement.

8.5.1 Motivation

Composing the ?-subgraphs of two components, as suggested in Section 8.4, corresponds to making a global refinement, i.e. refining *all* the possible causes for the indefinite result. We now show how to use abstraction in order to make the refinement more local and gradual by eliminating *one* of the causes for the indefinite result at a time.

Suppose that the coloring of the game graph G_1 for the lifted concrete component $M_1\!\uparrow$ results in an indefinite result. The coloring algorithms of Chapters 4 and 5 accompany such a result with a *failure state* and a *failure cause*, which is either a literal whose value in the failure state is $\bot$, or an outgoing may transition of the failure state in the underlying model which is not a must transition. Rather than globally refining the ?-subgraph, we wish to eliminate the failure cause returned by the coloring algorithm for $M_1\!\uparrow$. Suppose that s is the failure state. It abstracts all the states of $M_1 \| M_2$ that consist of s and a matching state of M_2. Eliminating the cause for failure amounts to exposing from M_2 the information that involves the failure, namely, which of the states composable with s satisfy the failure literal or have the corresponding transition and which of them do not, and splitting s accordingly. For example, in Figure 8.2, a possible failure cause in G_1 is the may transition of $M_1\!\uparrow$ from s_1 to s_2. In order to either remove it or turn it into a must transition, we need to consider all the states of M_2 which are composable with s_1. These are the states labeled r. We need to find out which of them have a transition to a state labeled r (i.e., a state composable with s_2), and which of them do not.

Clearly, the complete composition of the ?-subgraphs achieves this goal. However, it exposes more information than relevant for the given failure cause: It exposes *all* the information relevant to any possible cause for failure. In the general case, however, eliminating all failure causes is not necessary. Thus we do not want to resort to that (in this example it is indeed necessary, but in the general case not all the causes for failure need to be eliminated). We now sketch the idea that allows us to only consider the information from M_2 that is needed for eliminating the particular failure cause of $M_1\uparrow$. This will be described more formally in Section 8.5.2.

We abstract M_2 into $\hat{M}_2$. We start with a most coarse abstraction of M_2 w.r.t. $AP_1 \cap AP_2$, where each state is abstracted by its labeling, restricted to $AP_1 \cap AP_2$.

Definition 8.16. *Let $M_i = (AP_i, S_i, s_i^0, R_i, L_i)$ be a Kripke structure. The* most coarse abstraction *for M_i w.r.t. $AP' \subseteq AP_i$ is the KMTS $\hat{M}_i^* = (AP_i, 2^{AP'}, L_i(s_i^0) \cap AP', \emptyset, 2^{AP'} \times 2^{AP'}, L_i^*)$, where for $\hat{s} \in 2^{AP'}$, $L_i^*(\hat{s}) = \hat{s} \cup \{\neg p \mid p \in AP' \setminus \hat{s}\}$.*

Theorem 8.17. *$M_i \preceq \hat{M}_i^*$. The mixed simulation is given by $\{(s_i, L_i(s_i) \cap AP') \mid s_i \in S_i\}$.*

Proof. We show that $(s_i, L_i(s_i) \cap AP') \mid s_i \in S_i\}$ is a mixed simulation relation from M_i to $\hat{M}_i^*$. Clearly, the initial states are in H since the initial state of $\hat{M}_i^*$ is $L_i(s_i^0) \cap AP'$.

Let $(s_i, \hat{s}) \in H$, i.e. $\hat{s} = L_i(s_i) \cap AP'$. Then:

- $L_i(s_i) \supseteq L_i(s_i) \cap Lit' = L_i^*(\hat{s})$. The latter equality holds since by the definition of the most coarse abstraction, $L_i^*(\hat{s}) = \hat{s} \cup \{\neg p \mid p \in AP' \setminus \hat{s}\}$. $\hat{s} = L_i(s_i) \cap AP'$. Thus, $\{\neg p \mid p \in AP' \setminus \hat{s}\} = \{\neg p \mid p \in AP' \setminus (L_i(s_i) \cap AP')\} = \{\neg p \mid p \in AP'$ and $p \notin L_i(s_i)\} = \{\neg p \mid p \in AP'$ and $\neg p \in L_i(s_i)\} = L_i(s_i) \cap \neg AP'$. Therefore, altogether, the union of $\hat{s}$ and $\{\neg p \mid p \in AP' \setminus \hat{s}\}$ is equal to $L_i(s_i) \cap Lit'$.

- The requirement for must transitions is met vacuously as there are no must transitions in $\hat{M}_i^*$.

- The requirement for may transitions is also met trivially as every pair of states in $\hat{M}_i^*$ have a may transition between them.

□

An example of the most coarse abstraction of M_2 from Figure 8.2(a) w.r.t. $\{r\}$ appears in Figure 8.4. The construction of the most coarse abstraction requires almost no knowledge of the component. More precise transitions can be computed as described in Chapter 2. Starting from the most coarse abstraction of M_2, we iteratively model check the composition of M_1 and the abstract model $\hat{M}_2$. The model checking is performed in a compositional fashion, similarly to Section 8.4, without computing the full composition. If the result in some iteration is indefinite, we refine $\hat{M}_2$ depending on the failure cause over $M_1 \| \hat{M}_2$. The refinement of $\hat{M}_2$ is performed as described in Chapters 4 and 5 by splitting the abstract states in a way that eliminates the failure cause. Recall that our purpose was to eliminate a failure cause over $M_1\uparrow$. Since we start with a most coarse abstraction of M_2 w.r.t. the joint atomic propositions, $M_1 \| \hat{M}_2$ is initially isomorphic to $M_1\!\uparrow$. As a result, in the first iteration the failure cause over $M_1 \| \hat{M}_2$ reflects the failure cause over $M_1\!\uparrow$, and the refinement of $\hat{M}_2$ indeed exposes the relevant information from M_2. Similarly, in the next iterations, the failure cause over $M_1 \| \hat{M}_2$ reflects the failure cause over $M_1\!\uparrow$, after taking into consideration the elimination of previous failure causes. In this sense, in each iteration we eliminate one failure cause over $M_1\!\uparrow$, and $\hat{M}_2$ "accumulates" the information required to eliminate these failure causes.

This means that we keep one of the components, M_1, concrete, and construct an abstract environment for it, by applying an iterative abstraction-refinement on M_2, where refinement is aimed at eliminating the indefinite results that arise when considering M_1 with the abstract environment. An abstraction of M_2 (which comprises the environment of M_1) can be viewed as providing an assumption on M_2. From this point of view, when applying abstraction-refinement on M_2, the result is reminiscent of an iterative application of an asymmetric Assume-Guarantee rule. The next step is to make the approach symmetric by abstracting both components. This amounts to constructing abstract environments for both the components. In this case, refinement also needs to be applied on both components.

8.5.2 Compositional Abstraction-Refinement

We now describe in detail the combination of the compositional approach with abstraction-refinement. This provides a framework for using both the asymmetric and the symmetric abstraction-refinement approaches sketched above. On the one hand, we enhance the compositional model checking suggested in Section 8.4 by using abstraction and a more gradual refinement. On the other hand, we enhance the abstraction-refinement framework by making both the abstract model checking and the refinement compositional. We no longer require that the state spaces of the concrete components are finite, as long as the abstract state spaces are finite.

Compositional Abstraction Composition of abstract models (KMTSs) is defined in Definition 8.8. In order to ensure that the composition of two abstract models $\hat{M}_1 = (AP_1, \hat{S}_1, \hat{s}_1^0, R_1^+, R_1^-, \hat{L}_1)$ and $\hat{M}_2 = (AP_2, \hat{S}_2, \hat{s}_2^0, R_2^+, R_2^-, \hat{L}_2)$, for M_1 and M_2 respectively, results in an abstract model for $M_1 \| M_2$, we consider *appropriate* abstract models w.r.t. $AP_1 \cap AP_2$. We say that $\hat{M}_i$ is an *appropriate* abstract model of M_i w.r.t. $AP_1 \cap AP_2$ if $\hat{M}_i$ and M_i are related by a mixed simulation relation which is appropriate w.r.t. $AP_1 \cap AP_2$, as defined below.

Definition 8.18. *Let $H \subseteq S_i \times \hat{S}_i$ be a mixed simulation from M_i to $\hat{M}_i$, both defined over AP_i. We say that H is* appropriate *w.r.t. $AP' \subseteq AP_i$ if for every $(s_i, \hat{s}_i) \in H$, $L_i(s_i) \cap Lit' = \hat{L}_i(\hat{s}_i) \cap Lit'$, where Lit' denotes the set of literals over AP'.*

In particular, the most coarse abstraction w.r.t. $AP_1 \cap AP_2$ (see Definition 8.16) is appropriate w.r.t. $AP_1 \cap AP_2$. Appropriateness of $\hat{M}_1$ and $\hat{M}_2$ w.r.t. $AP_1 \cap AP_2$ means that the abstraction of each component only identifies states that agree on their labelings w.r.t. the joint atomic propositions. It ensures that if $(\hat{s}_1, \hat{s}_2)$ is a state of the abstract composition and $\hat{s}_1$ abstracts s_1 and $\hat{s}_2$ abstracts s_2, then since $\hat{s}_1$ and $\hat{s}_2$ agree on the joint labeling, then so do s_1 and s_2. This ensures that (s_1, s_2) is a state of the concrete composition, abstracted by $(\hat{s}_1, \hat{s}_2)$. We now have the following.

Theorem 8.19. *Let $\hat{M}_i$ be an appropriate abstract model for M_i w.r.t. $AP_1 \cap AP_2$ (for $i \in \{1,2\}$). Then $M_1 \| M_2 \preceq \hat{M}_1 \| \hat{M}_2$.*

Proof. Let $H_1 \subseteq S_1 \times \hat{S}_1$ and $H_2 \subseteq S_2 \times \hat{S}_2$ denote the mixed simulations between each component and its abstract model. Then the mixed simulation relation $H \subseteq S \times \hat{S}$ is given by $\{(s, \hat{s}) \mid (\pi_1(s), \pi_1(\hat{s})) \in H_1 \text{ and } (\pi_2(s), \pi_2(\hat{s})) \in H_2\}$. Clearly, the pair of initial states $((s_1^0, s_2^0), (\hat{s}_1^0, \hat{s}_2^0))$ is in H because $(s_1^0, \hat{s}_1^0)$ is in H_1 and $(s_2^0, \hat{s}_2^0)$ is in H_2 (since H_1 and H_2 are mixed simulations). Let $(s, \hat{s}) \in H$. Then

- $L(s) = L_1(\pi_1(s)) \cup L_2(\pi_2(s)) \supseteq \hat{L}_1(\pi_1(\hat{s})) \cup \hat{L}_2(\pi_2(\hat{s})) = \hat{L}(s)$ (the inclusion follows since H_1 and H_2 are mixed simulations).

- Let $(\hat{s}, \hat{s}') \in R^+$ in $\hat{M}_1 \| \hat{M}_2$. By the definition of (abstract) composition this implies that for each i, $(\pi_i(\hat{s}), \pi_i(\hat{s}')) \in R_i^+$. Then since H_1 and H_2 are mixed simulations, we conclude that for each i, there exists $t_i \in S_i$ s.t. $(\pi_i(s), t_i) \in R_i$ and $(t_i, \pi_i(\hat{s}')) \in H_i$. Since $\hat{M}_i$ is an appropriate abstract model for M_i w.r.t. $AP_1 \cap AP_2$, we have that $\hat{L}_i(\pi_i(\hat{s}')) \cap Lit_1 \cap Lit_2 = L_i(t_i) \cap Lit_1 \cap Lit_2$. Moreover, since $(\pi_1(\hat{s}'), \pi_2(\hat{s}'))$ is a state in $\hat{M}_1 \| \hat{M}_2$, then $\hat{L}_1(\pi_1(\hat{s}'))$ and $\hat{L}_2(\pi_2(\hat{s}'))$ agree on the joint atomic propositions, thus so do $L_1(t_1)$ and $L_2(t_2)$, and there exists a state (t_1, t_2) in $M_1 \| M_2$. In addition, by definition of H, $((t_1, t_2), \hat{s}') \in H$. It remains to show that $(s, (t_1, t_2)) \in R$. This is immediate from the definition of the (concrete) composition.

- Let $(s, s') \in R$. Then by definition of the (concrete) composition, there exists a transition $(\pi_i(s), \pi_i(s')) \in R_i$, and since H_i are mixed simulations, there exist $\hat{t}_i$ s.t. $(\pi_i(\hat{s}'), \hat{t}_i) \in R_i^-$ and $(\pi_i(s'), \hat{t}_i) \in H_i$. Since $\hat{M}_i$ is an appropriate abstract model for M_i w.r.t. $AP_1 \cap AP_2$, we have that $\hat{L}_i(\pi_i(\hat{s}')) \cap Lit_1 \cap Lit_2 = L_i(t_i) \cap Lit_1 \cap Lit_2$. Moreover, since $(\pi_1(s'), \pi_2(s'))$ is a state, then $L_1(\pi_1(s'))$ agrees with $L_2(\pi_2(s'))$ on the joint atomic propositions, thus so do $\hat{L}_1(\hat{t}_1)$ and $\hat{L}_2(\hat{t}_2)$, and there exists a state $(\hat{t}_1, \hat{t}_2)$. In addition, by definition of H, $(s', (\hat{t}_1, \hat{t}_2)) \in H$. It remains to show that $(\hat{s}, (\hat{t}_1, \hat{t}_2)) \in R^-$. This is immediate from the definition of the (abstract) composition.

□

Thus, if each of M_1 and M_2 is abstracted separately by an appropriate abstraction w.r.t. $AP_1 \cap AP_2$, then the composition of the corresponding abstract components $\hat{M}_1$ and $\hat{M}_2$ results in an abstract model for $M_1 \| M_2$. However, we do not wish to construct $\hat{M}_1 \| \hat{M}_2$ and model check it. Instead, we suggest to model check $\hat{M}_1 \| \hat{M}_2$ compositionally.

Compositional (abstract) Model Checking The general scheme is similar to the concrete case: we first try to make the most out of each (abstract) component separately, and if this does not result in a definite answer, we consider the product of the ?-subgraphs which enable to exchange information via a compact representation. We start by viewing each abstract component $\hat{M}_i$ as a partial model that abstracts their composition $\hat{M}_1 \| \hat{M}_2$.

Definition 8.20. *Let $\hat{M}_i = (AP_i, \hat{S}_i, \hat{s}_i^0, R_i^+, R_i^-, \hat{L}_i)$ be a KMTS. We lift $\hat{M}_i$ into a KMTS $\hat{M}_i\uparrow = (AP, \hat{S}_i, \hat{s}_i^0, R_i^+\uparrow, R_i^-\uparrow, \hat{L}_i\uparrow)$ over AP where $R_i^+\uparrow = \emptyset$, $R_i^-\uparrow = R_i^-$ and $\hat{L}_i\uparrow(\hat{s}) = \hat{L}_i(\hat{s})$.*

That is, when $\hat{M}_i$ is lifted into $\hat{M}_i\uparrow$, only the may transitions of $\hat{M}_i$ are useful, because must transitions are not really must w.r.t. $\hat{M}_1 \| \hat{M}_2$. Similarly to the concrete case:

Theorem 8.21. *$\hat{M}_1 \| \hat{M}_2 \preceq \hat{M}_i\uparrow$.*

Proof. Similar to the proof of Theorem 8.10. □

Corollary 8.22. *If $\hat{M}_i$ is an appropriate abstract model for M_i w.r.t. $AP_1 \cap AP_2$, then $M_1 \| M_2 \preceq \hat{M}_i\uparrow$.*

Proof. Follows immediately from Theorem 8.19 and Theorem 8.21 by the transitivity of $\preceq$. □

Therefore one can model check each of $\hat{M}_i\uparrow$ separately, and the definite results follow through to $M_1 \| M_2$. In fact, it is possible to show that $M_1 \| M_2 \preceq \hat{M}_i\uparrow$ holds even if we omit the appropriateness requirement. Thus appropriateness is not needed for this step. However, it is needed for the next steps, where we deduce from $\hat{M}_1 \| \hat{M}_2$ to $M_1 \| M_2$.

If both checks result in indefinite results, the (abstract) ?-subgraphs for both game graphs are produced and their product is considered. This is where the main difference between the concrete and the abstract case arises. Namely, having composed the ?-subgraphs of the two components resolves dependencies between them, but the result is still abstract, as it refers to the *abstract* composition $\hat{M}_1 \| \hat{M}_2$. This results in two differences compared to the concrete case.

First, the may edges do not necessarily become must edges. Instead, the distinction between may and must edges is determined by the type of the underlying transitions in the (unlifted) abstract models $\hat{M}_i$, which have been ignored so far. Second, it is now possible that a terminal vertex $v = (\hat{s}_1, \hat{s}_2) \vdash \psi$ in $G_\|$ with $\psi = l$ for a local literal $l \in Lit \setminus (Lit_1 \cap Lit_2)$ results from terminal vertices $\hat{s}_1 \vdash l$ and $\hat{s}_2 \vdash l$ which are *both* colored by ? in their subgraphs (one, since l is local to the other component, and is thus treated as indefinite, and the other due to the abstraction). We add this possibility as case (c) to Proposition 8.12 which characterizes the terminal vertices in the product graph $G_\|$. It is taken into account when determining the initial coloring of $G_\|$. Formally:

Definition 8.23 (Abstract Product Graph). *Let $G_{?1}$ and $G_{?2}$ be two abstract ?-subgraphs as above. Their product graph $G_\| = (V_\|, v_\|^0, E_\|^+, E_\|^-)$ is defined as before, except for the definition of $E_\|^+$: an edge $((\hat{s}_1, \hat{s}_2) \vdash \psi, (\hat{s}'_1, \hat{s}'_2) \vdash \psi')$ in $E_\|^-$ is also in $E_\|^+$ iff $\hat{s}_i R_i^+ \hat{s}'_i$ for each $i \in \{1, 2\}$.*

Proposition 8.24. *Let $v = (\hat{s}_1, \hat{s}_2) \vdash \psi$ be a terminal vertex in an abstract product graph $G_\|$. Then one of the following holds: (a) at least one of $\hat{s}_1 \vdash \psi$ and $\hat{s}_2 \vdash \psi$ is a terminal vertex in its subgraph and is colored by a definite color by the initial coloring of its subgraph. We denote this color by $col(v)$; Or (b) both $\hat{s}_1 \vdash \psi$ and $\hat{s}_2 \vdash \psi$ are non-terminal vertices but no outgoing edges were left in their composition; Or (c) $\psi = l$ for $l \in Lit \setminus (Lit_1 \cap Lit_2)$ and $\hat{s}_1 \vdash l$ and $\hat{s}_2 \vdash l$ are both terminal vertices colored by ? in their subgraphs.*

Definition 8.25. *We define the initial coloring function of an abstract product graph $G_{\|}$ as before (see Definition 8.13), with the addition that a terminal vertex that fulfills case (c) in Proposition 8.24 is colored ?.*

Theorem 8.26. *The resulting abstract product graph $G_{\|}$ is a closed subgraph of the game graph over $\hat{M}_1 \| \hat{M}_2$. In addition, the initial coloring function is correct and defined over all the terminal vertices in the subgraph.*

Proof. Similar to the proof of Theorem 8.14, with the following additions. First, in $\Box$ and $\Diamond$ vertices, the types of the outgoing edges (may vs. must) are determined by the same guidelines as the types of the transitions in $\hat{M}_1 \| \hat{M}_2$, which ensures that the abstract product graph is a subgraph of the game graph that corresponds to $\hat{M}_1 \| \hat{M}_2$. Second, for terminal vertices that are colored based on case (c) the correctness of the initial coloring function results from the fact that case (c) represents the case of a vertex $(\hat{s}_1, \hat{s}_2) \vdash l$ in $G_{\|}$, where $l \in Lit_i$ is local to $\hat{M}_i$ but its value in $\hat{s}_i$ is indefinite. This implies that its value in the corresponding state $(\hat{s}_1, \hat{s}_2)$ of $\hat{M}_1 \| \hat{M}_2$ is also indefinite (by the definition of composition), which makes the initial coloring correct. $\Box$

Along with Theorem 8.5, this implies that $G_{\|}$ can be colored correctly (w.r.t. the model checking of φ on $\hat{M}_1 \| \hat{M}_2$) using the 3-valued algorithm. If the initial vertex is colored by a definite color, then by Theorem 8.19 the result holds in $M_1 \| M_2$ as well and we are done.

Compositional Refinement Since an abstraction is used, the result of the model checking can be $\perp$, in which case the coloring algorithms of Chapters 4 and 5 return a failure cause that needs to be eliminated. The failure cause is either a literal whose value in a certain state is $\perp$, or a may transition of the underlying model which is not a must transition.

In our setting, the refinement step is done compositionally: If the failure cause is a literal l whose value in the failure state of $\hat{M}_1 \| \hat{M}_2$ is $\perp$, then l has to be a local literal of one of the components. This is because the abstraction is appropriate w.r.t. $AP_1 \cap AP_2$, which implies that no indefinite values for the joint atomic propositions occur in $\hat{M}_1 \| \hat{M}_2$. Thus, refinement need only be applied on the corresponding component.

Otherwise, the failure cause is a may transition (which is not a must transition) of $\hat{M}_1 \| \hat{M}_2$ that needs to be refined in order to result in a must transition or no transition at all. Let $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$ be this may transition. Then it results from may transitions $(\hat{s}_1, \hat{s}'_1)$ and $(\hat{s}_2, \hat{s}'_2)$ of $\hat{M}_1$ and $\hat{M}_2$ resp., such that at least one of them is not a must transition. In order to refine $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$, one needs to refine the individual may transitions in each component separately. If both of them are not must transitions, then refinement should be applied in each component. This is because a must transition in the composition results from must transitions in *both* components. Otherwise, refinement should only be applied in the component where it is not a must transition.

In each component where refinement is necessary, the refinement can be done as in Chapters 4 and 5. Moreover, in each component we adopt the incremental approach of Chapters 4 and 5 and avoid unnecessary refinement. In this approach, only vertices with indefinite colors are refined. In our setting, this corresponds to the ?-subgraph of each component. The result is the compositional abstraction-refinement loop presented in Figure 8.3.

Step 0 For $i \in \{1,2\}$, abstract M_i into $\hat{M}_i$ appropriately w.r.t. $AP_1 \cap AP_2$ (e.g. as in Definition 8.16).

Step 1 Model check each $\hat{M}_i\!\uparrow$ separately (for $i \in \{1,2\}$):

1. Construct the game graph G_i for φ and $\hat{M}_i\!\uparrow$.
2. Apply the 3-valued coloring on G_i. Let χ_i be the resulting coloring function.

If $\chi_1(v_1^0)$ or $\chi_2(v_2^0)$ is definite, return the corresponding model checking result for $M_1 \| M_2$.

Step 2 Consider the composition $\hat{M}_1 \| \hat{M}_2$:

1. Construct the ?-subgraphs for G_1 and G_2.
2. Construct the (abstract) product graph $G_\|$ of the ?-subgraphs.
3. Apply the 3-valued coloring on $G_\|$ (with the initial coloring function).

If $\chi_\|(v_\|^0)$ is definite, return the corresponding model checking result for $M_1 \| M_2$.

Step 3 Refine: Consider the failure cause returned by the coloring of $G_\|$ (where $\chi_\|(v_\|^0) = ?$).

If it is $l \in Lit_i$ then refine $\hat{M}_i$;

Else let it be the may transition $((\hat{s}_1, \hat{s}_2), (\hat{s}'_1, \hat{s}'_2))$. Then:

1. If $(\hat{s}_1, \hat{s}'_1)$ is not a must transition in $\hat{M}_1$, refine $\hat{M}_1$.
2. If $(\hat{s}_2, \hat{s}'_2)$ is not a must transition in $\hat{M}_2$, refine $\hat{M}_2$.

Refine the ?-subgraphs of G_1 and G_2 accordingly (as in the incremental approach);
Go to Step 1(2) with the refined subgraphs.

Figure 8.3: Compositional abstraction-refinement algorithm.

Note that the must transitions of each abstract component are only used when $G_\|$ is constructed. Thus, their computation can be deferred to step 2 and be limited to must transitions that are needed during model checking. Hyper-transitions can also be used, e.g. with the algorithm of Chapter 6 or Chapter 7.

Using the compositional abstraction-refinement starting from the most coarse abstraction w.r.t. $AP_1 \cap AP_2$ of one or both of the components results in the asymmetric, resp. symmetric, approach described in Section 8.5.1.

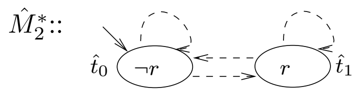


Figure 8.4: Most coarse abstraction of M_2 from Figure 8.2(a) w.r.t. $\{r\}$.

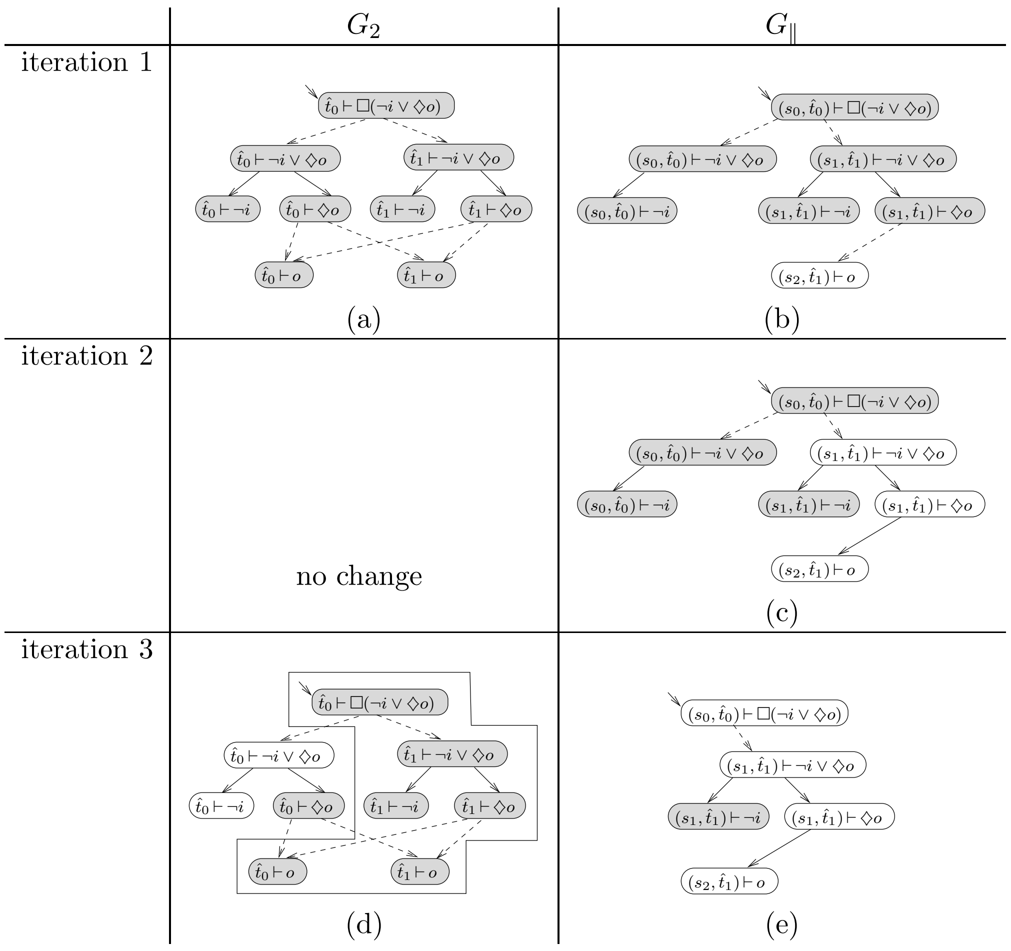
Theorem 8.27. *For finite concrete components, iterating the compositional abstraction-refinement process is guaranteed to terminate with a definite answer.*

Example 8.28. We demonstrate the compositional abstraction-refinement scheme on Example 8.15, which also served as a motivating example for introducing abstraction (see Section 8.5.1). We use the asymmetric approach, where only one component, in our case M_2, is abstracted. Initially, M_2 is abstracted using the most coarse abstraction w.r.t. the only shared atomic proposition r, as depicted in Figure 8.4. Namely, $\hat{M}_2 = \hat{M}_2^*$. The (initial) state $\hat{t}_0$ of the most coarse abstraction, labeled $\neg r$, represents the initial concrete state t_0 which is also labeled $\neg r$, and the state $\hat{t}_1$, labeled r, represents the two concrete states t_1 and t_2 which are labeled r. $\hat{M}_2$ is lifted into $\hat{M}_2\!\uparrow$ and model checked. The game graph G_2 of $\hat{M}_2\!\uparrow$ is depicted in Figure 8.5(a), where the initial vertex, as well as all other vertices, are colored ?. This is expected since the most coarse abstraction basically reveals no information on the concrete component.

Since all vertices are colored ?, the ?-subgraph of G_2 consists of the entire game graph. Recall that the initial vertex of the game graph G_1 that corresponds to $M_1\!\uparrow$ is also colored ? (see Figure 8.2(b)). Therefore, the product graph $G_\|$ of the ?-subgraphs of G_1 (see Figure 8.2(b)) and G_2 is constructed, as depicted in Figure 8.5(b). Note that $G_\|$ is still abstract: it contains may edges and terminal vertices which are colored ?. In fact, it is isomorphic to the ?-subgraph of G_1. Thus, its coloring is identical to that of G_1. In particular, it results in an indefinite value, accompanied with failure information.

Suppose that the failure state returned by the coloring is $(s_1, \hat{t}_1)$ with the failure cause being its outgoing may transition to the state $(s_2, \hat{t}_1)$, which is the underlying transition of the may edge from $(s_1, \hat{t}_1) \vdash \Diamond o$ to $(s_2, \hat{t}_1) \vdash o$ in the product graph. This may transition results from the concrete transition from s_1 to s_2 in M_1, and the self may loop of $\hat{t}_1$ in $\hat{M}_2$. Therefore, refinement is needed (only) in $\hat{M}_2$, where the corresponding transition is not a must transition. Note that while the refinement is performed in $\hat{M}_2$, it is in fact based on the point where more information is needed by the model checking instance of M_1, namely, whether or not the transition from s_1 to s_2 has a corresponding transition in M_2. This is because the failure cause is derived from the product graph which is isomorphic to the ?-subgraph of G_1. In particular, the may edge from $(s_1, \hat{t}_1) \vdash \Diamond o$ to $(s_2, \hat{t}_1) \vdash o$ in the product graph, from which the failure cause is derived, reflects the may edge from $s_1 \vdash \Diamond o$ to $s_2 \vdash o$ in G_1, which is one of the failure causes in G_1.

The refinement of $\hat{M}_2$ is aimed at splitting $\hat{t}_1$ such that each of the resulting substates either has a corresponding must transition or no transition at all. However, it turns out that in this case both of the concrete states represented by $\hat{t}_1$ have a corresponding transition, which means that the self loop of $\hat{t}_1$ can simply be added as a must transition and no split is required.



G_2
$G_\|$
iteration 1
iteration 2
iteration 3
no change
(a)
(b)
(c)
(d)
(e)
$\hat{t}_0 \vdash \Box(\neg i \vee \Diamond o)$
$\hat{t}_0 \vdash \neg i \vee \Diamond o$
$\hat{t}_1 \vdash \neg i \vee \Diamond o$
$\hat{t}_0 \vdash \neg i$
$\hat{t}_0 \vdash \Diamond o$
$\hat{t}_1 \vdash \neg i$
$\hat{t}_1 \vdash \Diamond o$
$\hat{t}_0 \vdash o$
$\hat{t}_1 \vdash o$
$(s_0,\hat{t}_0) \vdash \Box(\neg i \vee \Diamond o)$
$(s_0,\hat{t}_0) \vdash \neg i \vee \Diamond o$
$(s_1,\hat{t}_1) \vdash \neg i \vee \Diamond o$
$(s_0,\hat{t}_0) \vdash \neg i$
$(s_1,\hat{t}_1) \vdash \neg i$
$(s_1,\hat{t}_1) \vdash \Diamond o$
$(s_2,\hat{t}_1) \vdash o$

Figure 8.5: Game graphs arising during the run of the compositional abstraction-refinement algorithm described in Example 8.28.

The second iteration starts from the refined ?-subgraph of G_2 (which is the entire graph in this case). In fact, since no states were split, and since in the lifted component all transitions are viewed as may transitions, the so-called refined ?-subgraph remains unchanged and no recoloring is needed. In particular, the ?-subgraph computed in the second iteration remains the same, i.e. it consists of the entire game graph. However, when the product graph is constructed, the may edge from $(s_1, \hat{t}_1) \vdash \Diamond o$ to $(s_2, \hat{t}_1) \vdash o$ becomes a must edge, since it results from the concrete transition from s_1 to s_2 in M_1, and the self loop of $\hat{t}_1$ in $\hat{M}_2$ which turns out to be a must transition. As a result, the coloring changes as depicted in Figure 8.5(c). The initial vertex is still colored ?, and new failure information is provided.

Suppose that the failure state returned by the coloring is now $(s_0, \hat{t}_0)$ with the failure cause being the literal $\neg i$, whose value in $(s_0, \hat{t}_0)$ is indefinite (this failure information is derived from the terminal vertex $(s_0, \hat{t}_0) \vdash \neg i$ which is colored ? in the product graph). The literal $\neg i$ is local to M_2. Therefore, refinement is again needed in $\hat{M}_2$ only, and it reflects the fact that the information regarding $\neg i$ is needed by the model checking instance of M_1 (to resolve the ?-color of the vertex $s_0 \vdash \neg i$, which is one of the failure causes in G_1).

The refinement of $\hat{M_2}$ is aimed at splitting $\hat{t}_0$ such that each of the resulting substates is labeled by i or $\neg i$. However, it turns out that in this case the (only) concrete state represented by $\hat{t}_0$ is labeled $\neg i$, which means that the labeling of $\hat{t}_0$ can be updated and no split is required.

The third iteration starts from the refined ?-subgraph of G_2 from the second iteration. Recall that the ?-subgraph in the second iteration consists of the entire game graph. The only change in the refined ?-subgraph is that the terminal vertex $\hat{t}_0 \vdash \neg i$ is now colored T. As a result, the coloring of the refined ?-subgraph of G_2 changes as described in Figure 8.5(d), which also highlights the new ?-subgraph. Figure 8.5(e) presents the resulting product graph, and its coloring, where the initial vertex is now colored T, meaning that the property is verified.

While this small example does not really demonstrate the full power of abstraction, it nicely shows how the use of abstraction achieves the goal of a gradual refinement, where one failure cause is eliminated at a time. In this example, each refinement step reveals from M_2 additional information based on one failure cause found in the product graph, which reflects the model checking instance of M_1. This information is accumulated in $\hat{M_2}$.

Optimization In some cases, the ?-subgraphs can be pruned further at the end of an iteration, before they are refined, based on the product graph computed in the same iteration and its coloring. One possible reduction is the following. We say that a vertex v of a ?-subgraph *appears as a sub-vertex* of some vertex in the product graph if some vertex in the product graph shares the same subformula as v and its state (which is a pair of states of the individual components) consists of the state of v. Now, if a vertex v of a ?-subgraph does *not* appear as a sub-vertex of any vertex in the corresponding product graph, then v can be removed from its ?-subgraph as well. For example, at the end of the first iteration of Example 8.28, the vertices $\hat{t}_0 \vdash \Diamond o$ and $\hat{t}_0 \vdash o$ of the ?-subgraph depicted in Figure 8.5(a) can be removed since they are not sub-vertices of any vertex in the product graph of the first iteration, depicted in Figure 8.5(b).

Moreover, if a vertex v of a ?-subgraph only appears as a sub-vertex of vertices that are colored by T (resp. F) in the corresponding product graph, then v can be colored the same way in its ?-subgraph. For example, at the end of the first iteration of Example 8.28, the vertex $\hat{t}_1 \vdash o$ of the ?-subgraph depicted in Figure 8.5(a) can be colored T since it only appears as a sub-vertex of $(s_2, \hat{t}_1) \vdash o$, which is colored T, in the product graph of the first iteration, depicted in Figure 8.5(b). Furthermore, at the end of the second iteration, the vertices $\hat{t}_1 \vdash \neg i \vee \Diamond o$ and $\hat{t}_1 \vdash \Diamond o$ of the ?-subgraph from the second iteration, depicted in Figure 8.5(a), can be colored T since the corresponding vertices in the product graph of the second iteration, depicted in Figure 8.5(c), are all colored T. This allows for further pruning of the ?-subgraph which removes the entire subgraph whose root is the vertex $\hat{t}_1 \vdash \neg i \vee \Diamond o$, since once this vertex is colored T, it is no longer a witnessing son of the initial vertex.

Applying these optimizations in Example 8.28 allows us to prune the ?-subgraph depicted in Figure 8.5(a) at the end of the second iteration, before refinement is applied,

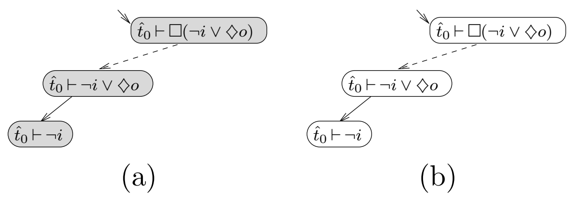

$\hat{t}_0 \vdash \Box(\neg i \vee \Diamond o)$
$\hat{t}_0 \vdash \neg i \vee \Diamond o$
$\hat{t}_0 \vdash \neg i$
(a)
$\hat{t}_0 \vdash \Box(\neg i \vee \Diamond o)$
$\hat{t}_0 \vdash \neg i \vee \Diamond o$
$\hat{t}_0 \vdash \neg i$
(b)

Figure 8.6: (a) Pruned ?-subgraph, and (b) its re-coloring after refinement which changes the color of $\hat{t}_0 \vdash \neg i$ to T.

into the one depicted in Figure 8.6(a). The refinement is then applied to this pruned subgraph. As before, it changes the color of the vertex $\hat{t}_0 \vdash \neg i$ to T (see Example 8.28). The third iteration starts from the resulting subgraph. Figure 8.6(b) exhibits its coloring, computed in the third iteration. The initial vertex is now colored T. Thus, due to these optimizations, the property is verified without the need to construct the product graph in the third iteration.

Relation to Assume-Guarantee Reasoning As explained in Section 8.5.1, an abstraction of a component M_i (which comprises the environment of the other component) can be viewed as providing an assumption on M_i. From this point of view, our compositional abstraction-refinement resembles iterative AG reasoning: When applying abstraction-refinement on one or both of the components, the result is an (asymmetric or symmetric) automatic mechanism for assumption generation. In each iteration, more information about the component is revealed based on the cause for the indefinite result. The use of conservative abstractions guarantees that the assumption describes the component correctly (by construction). Thus unlike typical AG reasoning, this need not be verified. Moreover, each iteration of our approach benefits from the compositional model checking described in the previous section.

8.6 Extension: Labeled Transition Systems

In this section we sketch how our compositional model checking technique can be applied to Labeled Transition Systems (LTSs). We refer to the case where the components are concrete. However, similar extensions are applicable in the abstract case. We follow the formulation of [23] for LTSs, but change the notation.

A Labeled Transition System (LTS) is a tuple $M = (A, S, s^0, R)$, where A is a set of observable actions, called the *alphabet* of M, S is a finite set of states, $s^0 \in S$ is the initial state, and $R \subseteq S \times (A \cup \{\tau\}) \times S$ is a transition relation. τ denotes a local action, unobservable to M's environment.

Composition of LTSs is defined as follows. Let $M_1 = (A_1, S_1, s_1^0, R_1)$ and $M_2 = (A_2, S_2, s_2^0, R_2)$ be two LTSs. Their composition, denoted $M_1 \| M_2$, is the LTS $M =$

(A, S, s^0, R), where $A = A_1 \cup A_2$, $S = S_1 \times S_2$, $s^0 = (s_1^0, s_2^0)$, and

$$\begin{aligned} R &= \{((s_1,s_2),a,(t_1,t_2)) \mid a \in A_1 \cap A_2 \text{ and } (s_1,a,t_1) \in R_1 \text{ and } (s_2,a,t_2) \in R_2\} \\ &\quad \cup\{((s_1,s_2),a,(t_1,s_2)) \mid a \in (A_1 \setminus A_2) \cup \{\tau\} \text{ and } (s_1,a,t_1) \in R_1 \text{ and } s_2 \in S_2\} \\ &\quad \cup\{((s_1,s_2),a,(s_1,t_2)) \mid a \in (A_2 \setminus A_1) \cup \{\tau\} \text{ and } (s_2,a,t_2) \in R_2 \text{ and } s_1 \in S_1\} \end{aligned}$$

Actions in $A_1 \cap A_2$ are *joint actions*, while actions in $(A_i \setminus A_{\bar{i}}) \cup \{\tau\}$ are *local actions* of M_i. The first type of transitions represent synchronization on the joint actions, whereas the second and third types represent interleaving of local actions, i.e. actions that do not belong to $A_1 \cap A_2$. In the latter type of transitions, one component proceeds along its transition, while the other component does not change its state.

It is convenient to add τ to the alphabet of the LTS, and distinguish between unobservable actions of different LTSs. As such, we redefine A_i to be $A_i \cup \{\tau_i\}$, and replace any occurrence of τ in R_i by τ_i.

Now, an LTS M_i can be viewed as an abstraction $M_i\uparrow$ of $M_1 \| M_2$ by viewing its local transitions (i.e., transitions which are labeled by local actions, including τ_i) as *must* transitions. This is because such transitions do not require synchronization, and thus their existence in the composition does not depend on the other component. The transitions which are labeled by joint actions are viewed as may transitions, as their existence in the composition depends on the other component. We also add to $M_i\uparrow$ additional idle may transitions $(s_i, \tilde{a}, s_i)$ for any $s_i \in S_i$, where $\tilde{a}$ is a new action that represents all the local actions of the other component, $M_{\bar{i}}$, and can therefore synchronize with all of them. The purpose of these idle transitions is to account for the local transitions of $M_{\bar{i}}$, which may or may not exists, and as such, might contribute transitions to the composition. Thus, $M_i\uparrow$ now contains both may and must transitions. The game graphs of the lifted LTSs are constructed and colored as before, and if necessary the ?-subgraphs are computed.

When the product graph of the ?-subgraphs of the lifted LTSs is constructed, every pair of vertices that share the same subformula is composed, with no further restriction on the states of the underlying LTSs. In practice, the vertices of the product graph are computed in a top-down fashion, hence only reachable pairs are included. The edges of the product graph are computed from the edges of the two ?-subgraphs by the same rules as the transitions of $M_1 \| M_2$. As a result, if for some vertex one ?-subgraph contains a may edge which is labeled by a joint action or $\tilde{a}$, but the other ?-subgraph has no matching edge, either since a corresponding transition does not exist in the corresponding component, or since it was pruned from the ?-subgraph, then the edge will not be included in the product graph. In addition, outgoing must edges of a vertex in one ?-subgraph which are labeled by local actions can be pruned as well: for such edges, the game graph of the other component always contains a corresponding (idle may) edge, however the corresponding edge can be pruned when the ?-subgraph is constructed (if it points to a non-witnessing son), in which case it will also be pruned from the product graph. This allows pruning of both may and must edges in the product graph. The remaining edges are all must edges (as they refer to the full system). Since the product graph is computed top-down, the pruning of the edges also results in pruning of the vertices.

The fact that there is no restriction in terms of the LTS states on which vertices of the ?-subgraphs are composed reflects the different synchronization mechanism of LTSs.

However, our approach can still yield reduction in this case as well, due to the fact that each of the ?-subgraphs is pruned. Thus, the resulting product graph is also pruned compared to the full game graph for $M_1 \| M_2$.

8.7 Concluding Remarks

This chapter suggests a new approach for compositional verification which utilizes the game-based techniques developed for 3-valued model checking of abstract models.

Our method is described for systems composed of two components, but it can be extended to the composition of n components. The main difference is that in the general case, after combining the ?-subgraphs of a subset of the (concrete) components, we will still be left with an abstract graph containing strict may edges and ?-colored terminal vertices due to information missing from the rest of the components.

The method can be adapted to take advantage of further knowledge of the system. For example, if the composed system has a total transition relation (e.g. [3, 30]), then our approach can be adapted to use an extended 3-valued semantics that yields definite results in more cases, based on the totality assumption. The extended semantics defines the value of a formula of the form $\Diamond\psi$ in an (abstract) state s to be tt also in the case where the value of ψ in *all* the may-successors of s is tt (dually for $\Box\psi$ when exchanging tt and ff). This is justified by the fact that the totality of the underlying concrete system ensures that at least one such successor exists.

We use KMTSs that have ordinary may and must transitions as abstract models, but the approach can be extended to handle hyper-transitions as well.

Chapter 9

Conclusion

In this research, we investigate abstraction-refinement and compositional techniques for specifications in the μ-calculus.

We extend the game-theoretic approach to μ-calculus model checking to the abstract case, using a 3-valued semantics, which enables to both verify and falsify properties on the same abstract model. For this purpose, we define a 3-valued model checking game. From the 3-valued model checking game, we derive two 3-valued model checking algorithms: one where a direct algorithm is used to solve the 3-valued game, and another where the 3-valued game is reduced to two 2-valued games. We accompany each of these algorithms with an automatic refinement, resulting in novel abstraction-refinement schemes for the full μ-calculus. Our abstraction-refinement schemes are incremental in the sense that in each iteration the model checking re-uses definite results from previous iterations.

Next, we study the precision of the abstract models that preserve the μ-calculus. We realize that previously suggested abstract models encounter a precision problem during refinement. Namely, formulas that had definite values (true or false) before refinement can become indefinite after refinement, which means that the refinement is not monotonic. To overcome this problem we suggest the use of hyper-transitions to under-approximate the concrete transitions.

Still in the context of precision, we realize that even when hyper-transitions are used to under-approximate the concrete transitions, the resulting abstract model is not as precise as the choice of the abstract states enables it to be. We formalize the notion of precision and suggest a new class of models where hyper-transitions are used both to under-approximate and over-approximate the concrete transitions. We prove that this class ensures maximal precision.

Hyper-transitions make the size of the abstract model potentially exponential in the number of abstract states. We therefore suggest efficient game-based model checking algorithms for abstract models with hyper-transitions that prevent the exponential blowup. Our first algorithm is suitable for models that use hyper-transitions only for the under-approximation of the concrete transitions. It is defined within an abstraction-refinement framework, where the idea is to "learn" hyper-transitions from the transitions of the previous iteration instead of computing all of them. Our second algorithm, on the other hand, can be used both independently and within an abstraction-refinement loop.

It handles alternation-free μ-calculus formulas and achieves maximal precision w.r.t. the checked property, while remaining quadratic in the number of abstract states.

Finally, we exploit the game-based techniques developed for 3-valued model checking to obtain a novel fully automatic compositional technique that can determine the truth value of the full μ-calculus w.r.t. a given system.

Examination of our ideas in practice is the subject of future work. For example, it would be interesting to compare the abstraction-refinement approaches of Chapters 4 and 5, where the refinement is *global*, i.e. applied on the entire abstract models, to our work in [35] which takes a *lazy* approach and applies the refinement more locally. Furthermore, each of these abstraction-refinement schemes uses a different mechanism for determining a criterion for refinement, which would be interesting to compare. Similarly, it would be interesting to compare our compositional technique to the automated assume-guarantee techniques suggested in the literature (e.g. [23, 5, 15]) for universal temporal logics.

Bibliography

[1] V. A. Aziz, T. R. Shiple and A. L. Sangiovanni-vincentelli. Formula-dependent equivalence for compositional CTL model checking. In David L. Dill, editor, *sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 324–337, Standford, California, USA, 1994. Springer-Verlag.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–109, New York, NY, USA, 2005. ACM Press.

[3] R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automating modular verification. In *CONCUR*, pages 82–97, 1999.

[4] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, Florida, October 1997.

[5] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided verification (CAV'05)*, volume 3576 of *LNCS*, pages 548–562, 2005.

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

[7] T. Ball and O. Kupferman. An abstraction-refinement framework for multi-agent systems. In *Proc. 21st IEEE Symp. on Logic in Computer Science*, 2006.

[8] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Copenhagen, Denmark, 2002.

[9] H. Barringer, D. Giannakopoulou, and C. Pasareanu. Proof rules for automated compositional verification through learning. In *2nd Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, 2003.

[10] O. Bernholtz, M. Vardi, and P.Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of the 6th International Conference on Computer Adided Verification (CAV'94)*, volume 818 of *LNCS*, pages 142–155. Springer-Verlag, 1994.

[11] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*, pages 274–287, 1999.

[12] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR'00*, volume 1877, pages 168–182, 2000.

[13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[14] S. Chaki, E. Clarke, O. Grumberg, J. Ouaknine, N. Sharygina, T. Touili, and H. Veith. State/event software verification for branching-time specifications. In *Proceedings of the 5th International Conference on Integrated Formal Methods (IFM)*, volume 3771 of *LNCS*, pages 53–69, 2005.

[15] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *Computer Aided verification (CAV'05)*, volume 3576 of *LNCS*, pages 534–547, 2005.

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

[17] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12:371–408, 2003.

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

[19] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT press, 1999.

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

[21] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Inf.*, 27:725–747, 1990.

[22] R. Cleaveland, P. Iyer, and D. Yankelevich. Optimality in abstraction of model checking. In *Static Analysis Symposium (SAS)*, pages 51–63, 1995.

[23] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 331–346, Warsaw, Poland, 2003.

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings*

of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL'77), pages 238–252, New York, NY, USA, 1977. ACM.

[25] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the Conference on Programming Language Implementation and Logic Programming (PLILP'92)*, pages 269–295. Springer-Verlag, August 1992. Lecture Notes in Computer Science 631.

[26] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.

[27] D. Dams and K. Namjoshi. The existence of finite abstractions for branching time model checking. In *19th IEEE Symposium on Logic in Computer Science (LICS)*, pages 335–344. IEEE Computer Society, 2004.

[28] D. Dams and K. S. Namjoshi. Automata as abstractions. In *6th international conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 216–232, Paris, France, 2005.

[29] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 170–179, 2004.

[30] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Detecting errors before reaching them. In *Computer Aided Verification*, pages 186–201, 2000.

[31] E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32th Symp. on Foundations of Computer Science (FOCS)*, pages 368–377. IEEE Computer Society Press, 1991.

[32] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396, 1993.

[33] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1986.

[34] A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008.

[35] H. Fecher and S. Shoham. Local abstraction-refinement for the mu-calculus. In *Proceedings of the 14th International SPIN Workshop on Model Checking Software*, pages 4–23, 2007.

[36] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE international conference on Automated software engineering (ASE)*, page 3, Washington, DC, USA, 2002. IEEE Computer Society.

[37] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR'01*, 2001.

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

[39] P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *Proceedings of VMCAI'2003 (4th Conference on Verification, Model Checking and Abstract Interpretation)*, volume 2575 of *LNCS*, pages 206–222, New York, 2003. Springer-Verlag.

[40] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. Don't know in the μ-calculus. In *6th international conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 233–249, Paris, France, 2005.

[41] O. Grumberg, M. Lange, M. Leucker, and S. Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Compuatation*, 205:1130–1148, August 2007.

[42] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[43] A. Gurfinkel, O. Wei, and M. Chechik. Systematic construction of abstractions for model-checking. In *VMCAI*, 2006.

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

[45] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *European Symposium on Programming (ESOP'01)*, volume 2028, pages 155–169, 2001.

[46] C. Jones. Specification and design of (parallel) programs. In *Information Processing 83: Proc. of the IFIP 9th World Congress*, pages 321–332, North-Holland, 1983.

[47] M. Jurdzinski. Small progress for solving parity games. In *STACS*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301, 2000.

[48] B. Konikowska and W. Penczek. Model checking for multi-valued computation tree logics. In *Beyond two: theory and applications of multiple-valued logic*, pages 193–210. Physica-Verlag GmbH, 2003.

[49] B. Konikowska and W. Penczek. On designated values in multi-valued CTL* model checking. *Fundamenta Informaticae*, 60(1–4):221–224, 2004.

[50] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.

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

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

[53] R. Küsters. Memoryless determinacy of parity games. In *Automata, Logics and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 95–106. Springer, 2002.

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

[55] K. Larsen and L. Xinxin. Equation solving using modal transition systems. In J. Mitchell, editor, *Proceedings of the Fifth Annual IEEE Symp. on Logic in Computer Science (LICS)*, pages 108–117. IEEE Computer Society Press, 1990.

[56] K. G. Larsen. Modal specifications. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science.* Springer-Verlag, June 1989.

[57] W. Lee, A. Pardo, J.-Y. Jang, G. D. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD*, pages 76–81, 1996.

[58] M. Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In *6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, 1999.

[59] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Autom. Softw. Eng.*, 12(3):349–382, 2005.

[60] J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In *Computer Aided Verification*, pages 316–327, 1999.

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

[62] D. Long, A. Browne, E. Clark, S. Jha, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *Computer Aided Verification, (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 338–350, 1994.

[63] K. S. Namjoshi. Abstraction for branching time properties. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 288–300, Boulder, CO, USA, 2003. Springer.

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

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

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

[67] C. S. Pasareanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Computer Aided Verification (CAV)*, pages 52–66, 2005.

[68] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*. Springer-Verlag, 1984.

[69] D. A. Schmidt. Closed and logical relations for over- and under-approximation of powersets. In *Static Analysis Symposium (SAS)*, pages 22–37, 2004.

[70] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 546–560, Barcelona, Spain, 2004.

[71] S. Shoham and O. Grumberg. Multi-valued model checking games. In *Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 3707 of *LNCS*, pages 354–369, 2005.

[72] S. Shoham and O. Grumberg. 3-valued abstraction: More precision at less cost. In *Twenty-First Annual IEEE Symposium on Logic In Computer Science (LICS)*, pages 399–410, Seattle, Washington, Aug. 2006.

[73] S. Shoham and O. Grumberg. Compositional verification and 3-valued abstractions join forces. In *Proceedings of the 14th International Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 69–86, Kongens Lyngby, Denmark, August 2007. Springer.

[74] S. Shoham and O. Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. *ACM Transactions on Computational Logic (TOCL)*, 9(1):1, 2007.

[75] C. Stirling. Local model checking games. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11, Berlin, Germany, 1995. Springer.

[76] C. Stirling. *Modal and Temporal Properties of Processes.* Springer, 2001.

[77] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[78] C. Stirling and D. J. Walker. Local model checking in the modal mu-calculus. In J. Diaz and F. Orejas, editors, *Proceedings of the 1989 International Joint Conference on Theory and Practice of Software Development*, volume 351–352 of *Lecture Notes in Computer Science*. Springer-Verlag, Mar. 1989.

[79] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5:285–309, 1955.

[80] G. Winskel. Model checking in the modal ν-calculus. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, 1989.

[81] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.

משתמשים. על כן, אנו מציעים משפחה חדשה של מודלים אבסטרקטיים שבה הקירוב מלמטה מתואר על ידי *היפר-קשתות*. היפר-קשת מחברת מצב מקור *לקבוצת* מצבי מטרה (במקום למצב מטרה יחיד). מודלים אלו מאפשרים עידון מונוטוני, אך סובלים מבעיית יעילות, כיוון שמספר ההיפר-קשתות עלול להיות אקספוננציאלי במספר המצבים. כפתרון לבעיית היעילות אנו מציעים סכמת אבסטרקציה-עידון חדשה המותאמת למודלים כאלו, ואשר בה היפר-קשתות מתווספות למודל באופן הדרגתי בכל איטרציה.

שימוש בהיפר-קשתות לצורך קירוב מלמטה פותר את בעיית האי-מונוטוניות, אך עדיין אינו מספק דיוק מקסימלי. אנו חוקרים את הדיוק של המודל האבסטרקטי ביחס לדיוק הטמון בבחירת המצבים האבסטרקטיים באופן בלתי תלוי במשפחת המודלים האבסטרקטיים בהם משתמשים, ומראים כי משפחות קיימות של מודלים אבסטרקטיים אינן מאפשרות דיוק מקסימלי. אנו מגדירים משפחה חדשה של מודלים אבסטרקטיים, בה היפר-קשתות משמשות הן לצורך קירוב מלמטה והן לצורך קירוב מלמעלה של המודל הקונקרטי, ומציעים בנייה של מודל אבסטרקטי מהמשפחה שהוא מדוייק ככל האפשר. כדי להתגבר על בעיית היעילות הנובעת מהשימוש בהיפר-קשתות, אנו מציעים אלגוריתם בדיקת מודל לתת לוגיקה של ה-μ-calculus, שבה אין קינון של נקודות שבת. האלגוריתם בונה את המודל האבסטרקטי *תוך כדי* בדיקת המודל ומחשב את ההיפר-קשתות *על פי הצורך*. בצורה כזו, אנו משיגים דיוק מקסימלי בסבוכיות ריבועית במספר המצבים האבסטרקטיים. לצורך השלמת התמונה, אנו משלבים את אלגוריתם בדיקת המודל האבסטרקטי בלולאת אבסטרקציה-עידון ומקבלים סכמת אבסטרקציה-עידון מונוטונית שבה בכל שלב מובטח דיוק מקסימלי ביחס למצבים האבסטרקטיים.

לסיום, אנו משתמשים בטכניקות האבסטרקציה-עידון החדשות על מנת לפתח אלגוריתם אימות מודולרי עבור ה-μ-calculus. במרבית המקרים אי אפשר לאמת את רכיבי המערכת בנפרד זה מזה. זאת כיוון שההתנהגות התקינה של כל רכיב תלויה באינטראקציה שלו עם הסביבה (שאר הרכיבים). על כן, הוצעה בספרות סכמת ה-Assume-Guarantee המציעה לאמת רכיב תחת *הנחה* על התנהגות הסביבה. לאחר מכן, הסביבה נבדקת על מנת *להבטיח* שהיא מספקת את ההנחה. רבות מהעבודות על אימות מודולרי מבוססות על סכמת ה-Assume-Guarantee ועל למידה לצורך ייצור הנחות. עבודות אלה עוסקות בלוגיקות אוניברסליות והרחבתן ללוגיקה כדוגמת ה-μ-calculus אינה ברורה.

אנו, לעומת זאת, מציעים אלגוריתם מודולרי המבוסס על טכניקות מבוססות משחקים שפותחו לצורך בדיקת מודל תלת-ערכית של מודלים אבסטרקטיים. אנו מציעים כיצד לזהות באופן אוטומטי ולהרכיב רק את החלקים של רכיבי המערכת שבהם ההרכבה אכן נחוצה לצורך בדיקת התכונה הרצויה. שאר החלקים מטופלים בנפרד. התוצאה היא טכניקה מודולרית אוטומטית לגמרי לקביעת ערך אמת של נוסחת μ-calculus במערכת נתונה. אנו משלבים את הטכניקה המודולרית עם אבסטרקציה כדי להקטין עוד יותר את רכיבי המערכת. לשם כך אנו מפתחים סכמת אבסטרקציה-עידון מודולרית שבבסיסה אלגוריתם בדיקת המודל המודולרי. סכמה זו מזכירה אלגוריתם Assume-Guarantee איטרטיבי עם מנגנון אוטמטי לייצור הנחות.

להיות מדומה, כך שאם המודל האבסטרקטי אינו מספק את הנוסחה, אי אפשר להסיק דבר על המודל הקונקרטי. הסמנטיקה *התלת-ערכית*, לעומת זאת, משמרת הן את ערך האמת true המעיד על הסתפקות, והן את ערך האמת false המעיד על הפרכה מהמודל האבסטרקטי למודל הקונקרטי. אולם, היא מוסיפה ערך אמת חדש: תחת הסמנטיקה התלת-ערכית יתכן כי ערך האמת של נוסחה במודל אבסטרקטי יהיה "לא מוחלט". במקרה זה, לא ניתן כל מידע על הערך במודל הקונקרטי. במילים אחרות, בעוד שאבסטרקציות מעל הסמנטיקה הדו-ערכית הן שמרניות ביחס לתוצאות חיוביות (הסתפקות) בלבד, הרי שאבסטרקציות מעל הסמנטיקה התלת-ערכית הן שמרניות ביחס לתוצאות חיוביות (הסתפקות) ושליליות (הפרכה) גם יחד. כתוצאה מכך, אבסטרקציות מעל הסמנטיקה התלת-ערכית מספקות תוצאות מדויקות לעיתים קרובות יותר הן עבור אימות והן עבור הפרכה.

סכמת האבסטרקציה-עידון המסורתית פותחה עבור לוגיקות אוניברסליות. היא מבוססת על אבסטרקציות דו-ערכיות, שבהן אם התוצאה היא false, אזי היא עשויה להיות מדומה. על כן, עידון דרוש כשתוצאת בדיקת המודל האבסטרקטי היא false ומטרתו של העידון היא למנוע קבלת תוצאות false. תוצאות כאלה מלוות פעמים רבות בדוגמאות נגדיות, לכן במרבית המקרים העידון מבוסס על ניתוח דוגמא נגדית שנמצאה בתהליך בדיקת המודל. גישה זו פחות מתאימה ללוגיקות כדוגמת ה- μ-calculus המשלבות הן אופרטורים אוניברסליים והן אופרטורים קיומיים, כיוון שכאשר מדובר בלוגיקות כאלה, המושג של דוגמא נגדית הוא פחות ברור. יתר על כן, סכמת האבסטרקציה-עידון המסורתית מבוססת על מודלים המקרבים את המודל הקונקרטי מלמעלה (לצורך אימות) או מלמטה (לצורך הפרכה). אולם, כאשר מעוניינים לשמר לוגיקות כדוגמת ה-μ-calculus, יש צורך במודלים אבסטרקטיים מורכבים יותר המספקים הן קירוב מלמעלה והן קירוב מלמטה של המודל הקונקרטי (בין אם משתמשים בסמנטיקה דו-ערכית, ובין אם משתמשים בסמנטיקה תלת-ערכית).

מחקר זה מפתח סכמות אבסטרקציה-עידון מבוססות משחקים עבור ה-μ-calculus. אנו משתמשים בסמנטיקה תלת-ערכית, המאפשרת הן אימות והן הפרכה של תכונות. עידון במקרה זה נחוץ רק כאשר תוצאת בדיקת המודל האבסטרקטי היא "לא מוחלט". אנו מציעים משחק בדיקת מודל אבסטרקטי בין שני שחקנים, מאמת ומפריך, כך שאם תוצאת בדיקת המודל היא true אזי למאמת יש אסטרטגיה מנצחת במשחק, אם תוצאת בדיקת המודל היא false אזי למפריך יש אסטרטגיה מנצחת במשחק, ואחרת תוצאת המשחק היא "תיקו". בדיקת המודל האבסטרקטי מבוצעת על ידי פתרון המשחק, כלומר קביעת השחקן המנצח. פתרון המשחק יכול להיעשות או על ידי אלגוריתם ישיר שאנו מציעים, או על ידי רדוקציה למשחק בדיקת מודל קונקרטי. לכל אחת מדרכי פתרון המשחק אנו מתאימים מנגנון עידון אוטומטי, במידה והתוצאה המתקבלת אינה מוחלטת.

אחד היתרונות של סכמות האבסטרקציה-עידון המתקבלות טמון בכך שהעידון מופעל רק על החלק של המודל שלגביו התקבלו תוצאות לא מוחלטות. כתוצאה מכך, המודל האבסטרקטי לא גדל שלא לצורך. יתר על כן, בדיקת המודל של המודל המעודן נעזרת בתוצאות מוחלטות שהתקבלו באיטרציות קודמות, דבר המוביל לבדיקת מודל *הדרגתית*. אלגוריתמי האבסטרקציה-עידון שאנו מציעים הם שלמים במובן שעבור מודלים קונקרטיים סופיים מובטח שהם יסתיימו עם תשובה מוחלטת.

אולם, כאשר מתבוננים על המבנה האבסטרקטי המתקבל לאחר העידון מתגלה תכונה בעייתית: מסתבר שתכונות שהיו בעלות ערכי אמת מוחלטים לפני העידון עשויות להפוך ללא מוחלטות לאחריו, כך שהעידון אינו *מונוטוני* מבחינת רמת הדיוק של המודל האבסטרקטי. בעיה זו נובעת מהדרך שבה מתואר הקירוב מלמטה במודלים האבסטרקטיים בהם

תקציר מורחב

מחקר זה עוסק באבסטרקציה-עידון ומודולריות בבדיקת מודל של הלוגיקה μ-calculus (תחשיב-μ).

בדיקת מודל היא גישה מצליחה לאימות תכונות של מערכות באופן אוטומטי. בהינתן מודל של מערכת ונוסחה בלוגיקה טמפורלית המתארת מפרט, בדיקת המודל מחזירה true אם המערכת מספקת את המפרט ו-false אחרת.

הלוגיקה בה אנו עוסקים במחקר זה היא ה-μ-calculus. לוגיקה זו מאפשרת לבטא תכונות מורכבות של מערכות ע״י שימוש באופרטורים מודאליים ובנקודות שבת. בעיות רבות בתחום האימות יכולות להיפתר ע״י תרגום לבדיקת מודל של נוסחאות כאלו.

החסרון המרכזי של פרוצדורת בדיקת המודל הוא *בעיית התפוצצות המצבים* ממנה היא סובלת. בעיה זו מתייחסת לדרישות הזכרון הגבוהות של בדיקת המודל הנובעות מכך שמודלים קונקרטיים (רגילים) של מערכות אמיתיות נוטים להיות גדולים מאוד. מרחב המצבים שלהם הוא אקספוננציאלי במספר המשתנים במערכת. מספר פתרונות הוצעו בספרות להתמודדות עם בעיית התפוצצות המצבים. שתיים מהגישות המבטיחות ביותר, בהן גם עוסק מחקר זה, הן אבסטרקציה ואימות מודולרי.

אבסטרקציה מסתירה חלק מפרטי המערכת, כך שהמודל האבסטרקטי המתקבל הוא קטן יותר. מטבע הדברים, בדיקת המודל של המודל האבסטרקטי מסתיימת לפעמים עם תשובה לא מוחלטת, שאינה מספקת מידע על ערך האמת במודל הקונקרטי. זהו סימן לכך שהאבסטרקציה הנוכחית אינה מספיק מפורטת על מנת לקבוע מה ערך האמת של התכונה הנבדקת במודל הקונקרטי, ולכן עליה לעבור *עידון*. עידון הוא תהליך הפוך לאבסטרקציה, שבו מוסיפים פרטים למודל האבסטרקטי. חזרה איטרטיבית על התהליך של אבסטרקציה, בדיקת מודל של המודל האבסטרקטי ועידון במידה ותוצאת בדיקת המודל האבסטרקטי לא מאפשרת להסיק מה התוצאה במודל הקונקרטי, נקראת *אבסטרקציה-עידון*.

ב*אימות מודולרי*, לעומת זאת, מנסים להתמודד עם בעיית התפוצצות המצבים על ידי בדיקת חלקים של המערכת בנפרד על מנת להימנע מבניית המודל של המערכת המלאה.

במחקר זה אנו מפתחים סכמות אבסטרקציה-עידון חדשות עבור ה-μ-calculus. אנו חוקרים הן את הדיוק והן את יעילות בדיקת המודל של מודלים אבסטרקטיים בהם משתמשים לצורך בדיקת תכונות ב- μ-calculus. לבסוף, אנו מרחיבים את עבודתנו בנושא אבסטרקציה-עידון לתחום של אימות מודולרי, ובכך מאחדים את כוחותיהן של שתי הגישות. להלן נפרט מעט יותר.

מודלים אבסטרקטיים מתוכננים בדרך כלל כך שיהיו שמרניים ביחס ללוגיקה נתונה. ישנן מספר סמנטיקות אפשריות לפירוש נוסחאות מעל מודלים אבסטרקטיים. הסמנטיקה *הדו-ערכית* מגדירה את ערך האמת של נוסחה במודל אבסטרקטי להיות true או false. כאשר ערך האמת הוא true, מובטח שהוא ישמר גם במודל הקונקרטי. אולם, ערך אמת false עלול

המחקר נעשה בהנחיית פרופ׳ ארנה גרימברג בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

אבסטרקציה-עידון ומודולריות בבדיקת מודל של תחשיב-μ

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

שרון שוהם

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

שבט תשס״ט חיפה פברואר 2009

אבסטרקציה-עידון ומודולריות בבדיקת מודל של תחשיב-μ

שרון שוהם