KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Reconstructing and Improving Parity Game Solvers with Justifications

Ruben Lapauw

Supervisors:
Prof. dr. Marc Denecker
Prof. dr. ir. Gerda Janssens

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

May 2021

Reconstructing and Improving Parity Game Solvers with Justifications

Ruben LAPAUW

Examination committee:
Em. prof. dr. ir. Patrick Wollants, chair
Prof. dr. Marc Denecker, supervisor
Prof. dr. ir. Gerda Janssens, supervisor
Prof. dr. ir. Frank Piessens
Prof. dr. ir. Tom Schrijvers
Prof. dr. Tom van Dijk
(University of Twente)
Prof. dr. Sven Schewe
(University of Liverpool)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

May 2021

© 2021 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Ruben Lapauw, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Acknowledgements

This text is the result of a journey in logic and games. And, like all PhD students, I did not travel alone.

First of all, I would like to thank Marc and Gerda. Without your support as my supervisors, I would not stand here today. Similarly, I thank Maurice, your guidance when writing and publishing papers was invaluable. I also thank the members of the jury, prof. Frank Piessens, prof. Tom Schrijvers, prof. Tom van Dijk, and prof. Sven Schewe. Your – surprisingly difficult – questions showed an understanding which boosted my joy for research. I am grateful to the IWT (agentschap voor Innovatie door Wetenschap en Technologie) and the KU Leuven for the financial and administrative support.

Thank you to all the colleagues I had the last seven years: Bart, Pieter, Jo, Joachim, Ingmar, Willem, Matthias, Laurent, Tim, Simon, Pierre, Djordje and Linde. I will remember not only our whiteboard discussions, emails... but also the KRR-weeks, the board games and the accepted-paper party's.

Finally, I want to thank my two children, Thibault and Clara, for keeping me sane while driving me insane during these unusual times. And I thank the rest of my family and my friends for their love and support.

In memory of Opa.

Abstract

In certain industries, such as the medicine or aviation, failure can result in unacceptable losses. To ensure safety, checks are implemented for every step in the process: standards, certifications, testing, maintenance schedules... For software, this encompasses researching diverse approaches including automatic testing, safer programming languages, declarative languages separating knowledge from execution, or generating provably correct software from a list of expected behaviours.

This thesis is situated in the domains exploring the two last ideas: Knowledge Representation and Reasoning (KRR) and Formal Verification (FV). KRR researches how knowledge can be represented for humans while supporting (automatic) reasoning with this knowledge. Formal Verification focuses on modelling and verifying the behaviour of systems.

One technique, used in KRR for improving reasoning capabilities of algorithms, is called justifications. Justifications store the derivation of facts in a graph. This derivation is then used for tracing unsupported arguments when invalidating assumptions. This text explores how justifications can improve parity game algorithms, which play a central role in FV. A parity game is an infinite two-player game on a directed graph. The core problem of a parity game is deciding the winner of certain nodes and a corresponding strategy. The solution of a parity game informs whether an given implementation is correct, whether an implementation is possible, or even allows extracting an valid implementation.

This text starts with adapting justifications to parity games. Next, we introduce an operation **Justify** to manipulate justifications for solving parity games. Then, four well-known recursive algorithms are reconstructed using this operation. These new variants are also implemented and evaluated experimentally and show justifications improve the state-of-the-art algorithms. Finally, we show that justifications for parity games are equivalent to justifications for propositional nested fixpoint definitions, an older logic with different applications.

Beknopte samenvatting

In sommige industrieën, zoals de medische wereld of de luchtvaart, kunnen storingen tot onaanvaardbaar verliezen leiden. Om de veiligheid te garanderen, zijn voor iedere stap in het proces controles ingebouwd, zoals standaarden, certificering, testen, onderhoud... Voor software omvat dit een waaier aan benaderingen, onder meer geautomatiseerd testen, veiligere programmeertalen, declaratieve talen die kennis van uitvoering scheiden, alsook het genereren van bewijsbaar correcte software.

Deze thesis bevindt zich in de domeinen die deze laatste twee ideeën verkennen: Kennis Representatie en Redeneren (KRR) en Formele Verificatie (FV). KRR onderzoekt hoe kennis kan voorgesteld worden met het oog op maximaal menselijk begrip terwijl het (geautomatiseerd) redeneren met deze kennis behouden blijft. Formele Verificatie focust op het modelleren van systeemgedrag en de verificatie van juist gedrag in alle scenario's.

Eén techniek die KRR gebruikt voor het versnellen van redeneertaken, is een justificatie ('justification'). Een justificatie, vaak voorgesteld als een grafe, bevat een logboek hoe gevolgen afgeleid worden. Op deze manier kunnen eenvoudig de ongesubstantieerde gevolgen gevonden worden wanneer veronderstellingen ongeldig blijken. Deze tekst verkent hoe het gebruik van justificaties oplossingsalgoritmen voor pariteitsspellen vereenvoudigt. Pariteitsspellen spelen een centrale role in Formele Verificatie. Het is een oneindig twee-speler spel gespeeld op een gerichte grafe. Het inherente probleem van een pariteitsspel bestaat uit het bepalen van de winnaars van bepaalde knopen alsook een bijhorende winnende strategie. Een oplossing van een pariteitsspel wordt gebruikt om te bepalen of een implementatie voldoet aan de regels, of zelfs voor het opstellen van een correcte implementatie.

Concreet wordt het gebruik van justificaties eerst aangepast voor pariteitsspellen: Een nieuwe vorm van justificaties wordt gedefinieerd samen met een passende operatie **Justify** om deze te manipuleren. Aan de hand van deze operatie

worden vier bekende recursieve algoritmen gereconstrueerd. Deze varianten werden geïmplementeerd en experimenteel geëvalueerd. De resultaten van deze experimenten tonen hoe het gebruik van justificaties de huidige beste oplossers versnelt. Tenslotte bewijzen we een equivalentie tussen pariteitsspellen en geneste vastepunts definities, een oudere logica met andere toepassingen.

List of Abbreviations

DefNF	Definition Normal Form	70
CTL	Computation Tree Logic	3
FO	First-Order Logic	2
FO(FD)	First Order Logic with Fixpoint Definitions	2, 67
FO(ID)	First-Order Logic with Inductive Definitions	2
FV	Formal Verification	1
ID	Inductive Definitions	2
KRR	Knowledge Representation and Reasoning	1
LTL	Linear Time Logic	3, 34
NFD	Nested Fixpoint Definition	67
PC(FD)	Propositional logic with Fixpoint Definitions	67
PG	Parity Game	9
PPG	Parametrised Parity Game	11
QD	Quasi Dominion	17
SAT	Satisfiability Problem	2
SAT(FD)	Satisfiability Problem of PC(FD)	67, 84
SAT(ID)	Satisfiability Problem with Inductive Definitions	23, 67
SCC	Strongly Connected Component	9
SMT	SAT Modulo Theories	67

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations vii

List of Symbols ix

Contents ix

List of Figures xiii

List of Tables xvii

1 Introduction 1

2 Preliminaries 7
2.1 Lattices 7
2.2 Graph notations 9
2.3 Parity games 9
2.4 Parametrised parity games 11
2.5 Existing parity game solving algorithms 13
2.5.1 Fixpoint iteration 13
2.5.2 Zielonka's algorithm 15
2.5.3 Priority promotion 17
2.5.4 Tangle learning 19
2.5.5 Other algorithms 21

3 Solving parity games with justifications 23
3.1 Introduction 23
3.2 Justifications 24

3.3 Operations on weakly winning justifications 26
3.4 Constructing winning justifications 27
3.4.1 The operation **Justify** 28
3.4.2 Using **Justify** to solve parity games 30
3.5 Justifying multiple nodes at once 33
3.6 Safely modifying parity games 35
3.7 Conclusion 37

4 Rebuilding recursive algorithms with justifications 39
4.1 Nested fixpoint iteration 40
4.1.1 A naïve algorithm 40
4.1.2 Optimisations 40
4.2 Zielonka's algorithm 43
4.3 Priority promotion 45
4.4 Tangle learning 47
4.5 Conclusion 56
4.5.1 Future work 57

5 Experimental evaluation of justification based algorithms 59
5.1 Introduction 59
5.2 Experimental setup 60
5.3 Results 61
5.4 Extended experimental results 65
5.5 Results split w.r.t. benchmark 67
5.6 Conclusion 69

6 An equivalence between propositional nested fixpoint definitions and parametrised parity games 71
6.1 A positive fragment of Propositional Logic 72
6.2 Nested fixpoint definitions 74
6.2.1 Syntax 74
6.2.2 Semantics 75
6.2.3 Witness justification 76
6.3 Solving fixpoint definitions by transformation 79
6.3.1 Transforming a nested fixpoint definition 79
6.3.2 Transforming parity game solutions to justification witnesses 81
6.4 Solving parity games by transformation 84
6.4.1 Transforming PPG's to nested fixpoint definitions . . . 84
6.4.2 Transforming witness justifications to PPG solutions . . 86
6.5 Conclusion 88
6.5.1 Future work 88

7 Conclusion 91

7.1 Contributions 91
7.2 Future work 92

Bibliography **95**

List of publications **105**

List of Figures

1.1 A reduced parity game. 4
1.2 The resulting Mealy machine with two states, alternating $\neg eat_A, eat_B$ and $eat_A, \neg eat_B$ regardless of the input of $hungry_A$ and $hungry_B$. 4

2.1 A parity game with nodes $a, \ldots, f$. Square nodes are owned by Even, diamonds by Odd. The label inside a node shows its name and priority. 10
2.2 The solution of Fig. 2.1: The label on top of a node is the winner. A bold edge belongs to the winning strategy of the owner of its origin node. A slim edge starts in a node that is lost by its owner. All other edges are dotted. 11
2.3 A parametrised parity game with nodes $a, \ldots, f$, $P_0 = \{d\}$ and $P_1 = \{a\}$, and winning strategies for 0 and 1. The two parameter nodes are in bold. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority; the label on top of a node is the winner. A bold edge belongs to a winning strategy (of the owner of its start node). A slim edge is one starting in a node that is lost by its owner. All remaining edges are dotted. 12
2.4 A parametrised parity game and strategy, after withdrawing d from the parameters, and adding e and f. 12

3.1 A parametrised parity game with nodes $a, \ldots, f$, $P_0 = \{d\}$ and $P_1 = \{a\}$, and winning strategies for 0 and 1. The two parameter nodes are in bold. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority; the label on top of a node is the winner. A bold edge belongs to a winning strategy (of the owner of its start node). A slim edge is one starting in a node that is lost by its owner. All remaining edges are dotted. 25
3.2 Above, in solid line the edges of the justification graph of the winning but unsafe justification of Figure 3.1 and below the result of justifying node a, a non-winning justification! The play $a, b, a, b, \ldots$ is won by Even while $H(a) = H(b) = 1$. 31
3.3 Above: The running example parity game with its solution. Below: A different parity game with a single change: the priority of node c is now 4. Re-parametrising only the affected node results in an unsafe justification. 36
3.4 Above: The running example parity game with its solution. Below: The same modified parity game where the priority of node c is now 4. Re-parametrising all nodes reaching the affected node maintains a safe justification. 37

4.1 Parity games for Example 6 41
4.2 A part of a parity game containing a tangle. Square nodes are owned by Even, diamonds by Odd, circles fall outside the scope of this example. The labels inside a node are the name, priority and, in bold, the current justification level. The figure shows a tangle with 4 nodes (b, c, f and g) and 6 edges. 48
4.3 A different justification for the same part of the parity game as Figure 4.2 in which the previous tangle is justifiable. 50
4.4 The result of justifying the tangle in Figure 4.3. 51

5.1 Cactusplot showing runtime of the slowest 100 parity games for the top six algorithms 63
5.2 Comparisons of run times 63
5.3 Comparisons of run times (extended results) 67

6.1 A corresponding witness justification for the nested definition in Example 9. Every literal is annotated with its type ('C' for conjunctive nodes, 'D' for disjunctive, 'O' for open) and with its evaluation (**t** and **f**). Direct justifications of conjunctive literals are solid while disjunctive literals use bold edges. Dotted edges are edges in the definition graph but not in this justification. . 78

6.2 A parity game with nodes $a, \ldots, f$. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority. 80
6.3 The parity game of Figure 6.2 modified for $I_O(f) = \mathbf{t}$. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority. Node f is won by Even. . . . 82
6.4 The transformed parity game of the running example with a winning justification, a copy of Figure 6.3 85
6.5 A corresponding witness justification for the nested definition in Example 12. Every literal is annotated with its type ('C' for conjunctive nodes, 'D' for disjunctive) and with its evaluation ($\mathbf{t}$ and $\mathbf{f}$). Direct justifications of conjunctive literals are solid while disjunctive literals use bold edges. Dotted edges are unused. . 87

List of Tables

5.1 Number of solved instances in less than 1, 10, and 100 seconds and par2 score (lower is better), sorted by the par2 score . . . 62
5.2 The reduction of runtime is coupled to a reduction in number of attracted nodes . . . 64
5.3 Number of instances solved by the additional algorithms in less than 1, 10, and 100 seconds and par2 score (lower is better), sorted by par2 score. . . . 66
5.4 PG-bench crafted runs . . . 68
5.5 Large random graphs . . . 69

Chapter 1

Introduction

Computers are now used in every domain, and by every industry and government. However, using and programming computer systems is inevitably linked with human errors. One error can affect millions of devices, may result in millions of dollars in lost revenue, or even billions of dollars in damage. In fact, Tricentis [Tri18] estimates computer errors result in losses of over $1.7 trillion every year. Thus, multiple fields within computer science research how to avoid, prevent, detect and fix such errors.

This thesis focuses on two related fields involved in reducing human errors: Knowledge Representation and Reasoning (KRR) [BL04], and Formal Verification (FV) [Cla+18]. KRR studies knowledge: how it is used, how to represent it, and how such representations can be used to solve problems. Formal Verification studies correctness of systems: with a formal language such systems can be represented and various properties can be proven.

Knowledge Representation and Reasoning

The domain of Knowledge Representation and Reasoning studies how humans express and use knowledge to automate problem-solving with computers. This consists of two steps: First, knowledge must be represented. To facilitate this, KRR devises *formal languages* and studies strengths and weaknesses for different problems. Second, expressed knowledge is used to automate tasks. Most formal languages include rules to reason with. With the rise of computers, these rules have been implemented in software, resulting in *automated reasoning*.

One such formal language is First-Order Logic with Inductive Definitions (FO(ID)) [DT08]. This language combines elements from two paradigms in Knowledge Representation: the language is based on the classic First-Order Logic (FO), and it is extended with Inductive Definitions (ID) from the rule-based approaches. This declarative language is supported by a software implementation called IDP. This program allows to represent knowledge declaratively, i.e., without assuming a predetermined problem to solve. An imperative shell around the language then allows running *inferences* - calculation tasks - on the knowledge to solve diverse problems. As the declarative language is based on logic, most inferences depend on solving the Satisfiability Problem (SAT) augmented with theories for arithmetic and inductive definitions, together with optimisations and heuristics. The fusion of multiple paradigms enhances the expressivity and reasoning desired from a KRR-language.

Justifications One of the prevailing improvements in IDP is the use of justifications [Mar+08] in its inductive definition modulo theory. This technique is based on research of Denecker and De Schreye [DD93]. Other research based on justifications include solving FO(FD) [HD09] (see Chapter 6), lazy grounding [DDS12; BW18], and relevance [Jan+16].

The idea of justifications consists of determining an explanation why a fact is true, allowing one to build a 'knowledge graph'. This additional knowledge is then used to improve different inferences. One improved inference is determining which derived information is invalidated when assumptions change. This inference then forms an algorithmic basis for maintaining valid knowledge graphs instead of restarting from scratch. Efficiently safeguarding large parts of the knowledge graph has been used to speed up different inference algorithms.

The original idea for this work was to improve the justification-based algorithm detailed in [HD09]. However, a way to evaluate the implementations was missing. In search of applications, we found a close relation with parity games used in Formal Verification.

Formal Verification

Formal Verification wields different languages with a single goal: being correct. After specifying the workings of systems and their expected properties, claims are verified. Another theme in Formal Verification is controller synthesis, in which software generates a verified system when given the required properties.

Finite systems, and infinite systems with finite representations, are generally represented by, or transformed to, finite state machines. Properties are often

expressed in boolean equation systems [Cra+13; KP12], SystemVerilog assertions, and different temporal logics [Wal96; GTW02] such as linear time logic (LTL), computational tree logic (CTL) and μ-calculus. Using software, the claimed properties can be verified automatically. Examples using this approach include verification of digital circuits in hardware, secure protocols in cryptography and robotic control systems.

Large and infinite systems are often infeasible or undecidable. To verify properties other solutions are required. Example solutions include automatic theorem proving (SPASS [Gae95], Prover9 [McC10]), interactive theorem proving (HOL, Isabelle, Coq, Adga) or reductions to SAT modulo theories (Boogie [GLM11], Z3 theorem prover [MB08], Yices [Dut14] ...).

This thesis focuses on the sub-domain that verifies finite systems. A lot of temporal properties of finite systems can be solved by transformation to parity games. In particular, we aim to improve solving parity games by applying justifications.

Parity games are two-player games played on a directed graph without leaves. A node has an owner (a player, either Even or Odd) and an integer priority. A play is an infinite path in the graph where the owner of a node chooses which outgoing edge to follow. A play and its nodes is won by Even if the highest priority that occurs infinitely often is even and by Odd otherwise. A parity game is solved when the winner of every node is determined and proven.

Logics such as Linear Time Logic (LTL) and Computation Tree Logic (CTL) are used to express properties of interacting systems. Synthesis consists of extracting an implementation with the desired properties. Typically, formulas in such logics are handled by reduction to other formalisms. LTL can be reduced to Büchi-automata [VW86; Kes+93], determinised with Safra's construction [Saf88], and transformed to parity games [Pit06]. Other modal logics have similar reductions: Kant and Pol [KP12] show how to reduce boolean equation systems, while CTL* can be reduced to automata [BVW94], to μ-calculus [CGR11], and recently to LTL-formulas [BSK17]. All are reducible to parity games which can be solved directly.

Quasi-polynomial time algorithms for solving parity games exist [Cal+17; Fea+17; Par19]. However, all current state-of-the-art algorithms (Zielonka's algorithm [Zie98], strategy-improvement [Sch08], priority promotion [BDM18; BDM16b; BDM16a] and tangle learning [van18a]) are exponential.

One of the tools that support the synthesis of implementations for such formulas is Strix [LMS20; MSL18], one of the winners of the SyntComp 2018 [Jac+19] and SyntComp 2019 competition. This program reduces LTL formula on the

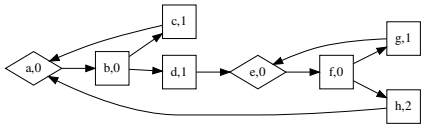


Figure 1.1: A reduced parity game.

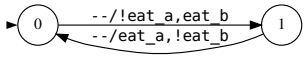


Figure 1.2: The resulting Mealy machine with two states, alternating $\neg eat_A, eat_B$ and $eat_A, \neg eat_B$ regardless of the input of $hungry_A$ and $hungry_B$.

fly to parity games. A game has three possible outcomes: (i) the parity game needs further expansion, (ii) the machine wins the game, i.e., an implementation is feasible, (iii) the environment wins, i.e., no implementation exists. Strix also extracts an implementation with the specified behaviour, e.g., as a Mealy machine.

Consider a formula based on the well-known dining philosophers problem:

$$\begin{array}{ll} G(hungry_A \Rightarrow F eat_A) \wedge & \text{If A is hungry, he will eventually eat} \\ G(hungry_B \Rightarrow F eat_B) \wedge & \text{If B is hungry, he will eventually eat} \\ G(\neg eat_A \vee \neg eat_B) & \text{A and B cannot eat at the same time.} \end{array} \tag{1.1}$$

Here $(G\phi)$ means ϕ holds in every future trace and $(F\phi)$ means ϕ holds in some future trace where a trace is a succession of states.

Strix transforms the LTL-formula 1.1 to the parity game of Figure 1.1. The machine (Even) plays in the square nodes and the environment (Odd) in the diamond nodes. By playing in state b to d and in state f to h Even wins every node as 2 is then the highest priority that occurs infinitely often in every play. From the solution, Strix extracts a 2-state Mealy machine (Figure 1.2). Its behaviour satisfies Formula 1.1: both philosophers alternate eating regardless of their hunger.

Outline and contributions

With this text, we contribute the insights on how justifications can be used to rebuild different parity game solving algorithms. The core focuses on a justification-based framework for solving parity games in three parts: first, the framework is specified and proven, second, we use the framework to rebuild well-known algorithms, and third, we show these rebuilt variants improve the state-of-the-art for solving parity games. The main matter is preceded by the preliminaries and is followed by an additional chapter linking parity games back to previous applications of justifications and by a conclusion.

- The preliminaries, **Chapter 2**, introduce the necessary concepts used in the rest of this thesis. This includes parity games and an extension, parametrised parity games. This chapter also discusses different well-known parity game solving algorithms.
- **Chapter 3** details a justification-based structure and an operation **Justify**. Proving the invariants of this operation shows repeated applications of **Justify** eventually solve parity games. This yields a non-deterministic algorithm for solving parity games.
- **Chapter 4** shows different algorithms correspond to different executions of the non-deterministic algorithm: variants of fixpoint induction, Zielonka's algorithm, priority promotion, and tangle learning are rebuilt based on the **Justify**-operation. Rebuilding these algorithms in the same framework allows to compare the strengths and weaknesses of these algorithms starting from the same basis and to explain general performance differences.
- An experimental validation is performed in **Chapter 5**. One of the core properties of a justification is identifying all nodes that violate the invariants in order to reset them. In Chapter 4 it was noted the justification variants reset a subset of the nodes compared to the original implementations. Some of the algorithms described of that chapter were implemented and benchmarked. In this chapter the results are discussed and the expected practical improvement is verified.
- Finally, **Chapter 6** looks back to the origins of justifications: this chapter shows an equivalence between parity games and the nested fixpoint definitions defined by Hou et al. [HDD10]. This chapter serves as a set-up for exchanging additional techniques and applications.
- **Chapter 7** summarises the conclusions of the previous chapters and speculates about future improvements and applications.

Chapter 2

Preliminaries

This chapter contains preliminary information for the later chapters. Of note is the introduction of parity games in Section 2.3 and its extension to parametrised parity games in Section 2.4. Finally, Section 2.5 discusses various well-known algorithms for solving parity games.

Section 2.1 up to Section 2.3 and Section 2.5.2 up to Section 2.5.4 have been published in [LBD20]. Section 2.4 will be published in [LBDar].

2.1 Lattices

Least- and greatest-fixpoints

Some parts of this thesis use functions over complete lattices. If these functions are monotonic, they have a unique and well-defined fixpoint. A partial order $\leq$ over a set L is a lattice $(L, \leq)$ if every two elements of L have an infimum and supremum. A lattice is complete if there exits a minimal element $\bot$, and a maximal element $\top$. A chain in a lattice is a sequence of lattice elements $l_1, l_2, \cdots \in L$ for which $l_1 \leq l_2 \leq \ldots$. The length of a finite chain $l_1 \leq l_2 \leq \cdots \leq l_n$ is n. If all chains in a lattice are finite then the lattice has a height equal to the longest chain. For a finite lattice, the height is less than or equal to the number of elements.

Given a lattice $(L \leq)$, a function $f : L \to L$ is monotonic if for all $x, y \in L$ for which $x \leq y$: $f(x) \leq f(y)$. Tarski [Tar55] proved that, for complete lattices L,

monotonic functions $f : L \to L$ have a unique least fixpoint $lfp(f)$ and a unique greatest fixpoint $gfp(f)$.

Theorem 2.1 (Monotonic fixpoint [Tar55]). *Let $L, \leq$ be a complete lattice and let f be a monotonic function from L to L. Denote with $f^i(l)$ the result of iterating f on the element $l \in L$: $f^0(l) = l$ and $f^i(l) = f(f^{i-1}(l))$. The least-fixpoint of f is the supremum of the elements in the ascending Kleene chain:*

$$\bot \leq f(\bot) \leq f^2(\bot) \leq \cdots \leq f^n(\bot) \leq \ldots$$

We denote $lfp(f)$ as $\sup\{f^n(\bot) | n \in \mathbb{N}\}$.
The greatest-fixpoint of f is the infimum of the elements in the descending Kleene chain:

$$\top \geq f(\top) \geq \cdots \geq f^n(\top) \geq \ldots$$

We denote $gfp(f)$ as $\inf\{f^n(\top) | n \in \mathbb{N}\}$.

If the lattice has height h then $lfp(f) = f^h(\bot)$ and $gfp(f) = f^h(\top)$.

μ-calculus

In 1969, De Bakker and Scott [DS69] introduced modal μ-calculus, which adds least- and greatest-fixpoints to propositional modal logic. As Seidl [Sei96] states, μ-calculus is essentially the fixpoint evaluation of functions monotonic in multiple arguments: let $(\mathcal{S}, \leq)$ be a lattice, let $\Phi : \mathcal{S}^n \to \mathcal{S}$, with $n \geq 2$, be a function that is monotone in all arguments. Then for fixed $S_n, \ldots, S_2 \in \mathcal{S}$ the function $\phi_{(S_n,\ldots,S_2)} : S_1 \to \Phi(S_n, \ldots, S_2, S_1)$ is monotone. Thus the functions $\Psi_\mu : (S_n, \ldots, S_2) \mapsto lfp(\phi_{(S_n,\ldots,S_2)})$ and $\Psi_\nu : (S_n, \ldots, S_2) \mapsto gfp(\phi_{(S_n,\ldots,S_2)})$ are well-defined. Moreover they are monotone in all arguments; their least and greatest fixpoints are well-defined. For ease of notation, De Bakker and Scott use a syntax based on lambda-calculus: $\mu S_1.\Phi(S_n, \ldots, S_2, S_1)$ is a function calculating a least-fixpoint, $\nu S_1.\Phi(S_n, \ldots, S_2, S_1)$ calculates the greatest-fixpoint.

For a finite set T, $(2^T, \subseteq)$ is a lattice with height $|T|$. If the function $\Phi : \left(2^T\right)^n \to 2^T$ is monotonic and computable, the fixpoints can be calculated with an algorithm by Emerson et al. [EL86]. For example, the expression $\nu S_n.\mu S_{n-1}. \ldots \nu S_2.\mu S_1.\Phi(S_n, \ldots, S_1)$ has a unique and well-defined fixpoint. Implementing previous fixpoint expression naïvely evaluates Φ, in the worst case, $|T|^n$ times. An improved algorithm by Long et al. [Lon+94] reduces the number of evaluations to $|T|^{\lceil n/2 \rceil}$. A similar reduction was found by Seidl [Sei96].

2.2 Graph notations

A directed graph G is a pair (V, E) of a set of nodes V and a set of edges $E \subseteq V \times V$. A pair $(x, y) \in E$ is an edge from node x to node y. The edge is an outgoing edge of node x and incoming for y. We call the node x the origin and the node y the endpoint.

Consider a directed graph G. With $Out_G(v)$ we denote the set of outgoing edges from v. We denote $G{\downarrow_v}$ as the set of nodes reaching v in the directed graph G. We denote $G{\uparrow_v}$ as the set of nodes reachable from v in the directed graph G.

A directed graph is strongly connected iff every node in the graph reaches every other node. The set of strongly connected components (SCC) of a directed graph is a partition in maximal strongly connected sub-graphs. The SCC a node v belongs to can be calculated as $G{\downarrow_v} \cap G{\uparrow_v}$; every node in this set reaches every other node through v. Tarjan's algorithm [Tar72] is a well-known algorithm for efficiently partitioning a graph in SCC's. This algorithm has time complexity $\Theta(|V| + |E|)$ for a graph (V, E).

2.3 Parity games

A parity game [Wal96] is a two player game of player 0 (Even) against 1 (Odd). We use $\alpha \in \{0, 1\}$ to denote a player and $\bar{\alpha}$ to denote its opponent. Formally, we define a *parity game* (PG) as a tuple $\mathcal{PG} = (V, E, O, Pr)$ with V a finite set of nodes, $E \subseteq V \times V$ a set of directed edges, $O : V \mapsto \{0, 1\}$ the owner function, and Pr the priority function $V \to \mathbb{N}$ mapping nodes to their priority; (V, E) is also called the game graph. Each $v \in V$ has at least one possible move; for all nodes $v \in V$ there exists an edge (v, w) in E. We use O_α to denote the nodes of which α is the owner. We call the player $\alpha = (p \mod 2)$ the *winner of priority* p for the reason explained below. Given a set of nodes S, a player α, a priority i and a relation $\sim$ in the set $\{=, \neq, \leq, \geq, <, >, \equiv_2, \not\equiv_2\}$, $S^{\sim i}$ denotes the set of nodes $\{v \in S | Pr(v) \sim i\}$. E.g., $V^{<i}$ is $\{v \in V | Pr(v) < i\}$ and $V^{\equiv_2 0}$ is the set of nodes with a priority which is equivalent – modulo 2 – with priority 0, i.e., the nodes with an even priority.

A *play* (in node v_1) of the parity game is an infinite sequence of nodes $\langle v_1, v_2, \ldots, v_n \ldots \rangle$ of moves along the game graph: $\forall i : v_i \in V \wedge (v_i, v_{i+1}) \in E$. We use π as a mathematical variable to denote a play. $\pi(i)$ is the i-th node v_i of π. In a play π, it is the owner of the node v_i that decides the move (v_i, v_{i+1}). There exists plays in every node. The winner of a play is the winner of the

highest priority through which the play passes infinitely often. Formally:

$$Winner(\pi) = \limsup_{i \to +\infty} \{Pr(\pi(j)) | j \geq i\} \mod 2.$$

The key questions for a parity game $\mathcal{PG}$ are, for each node v: Who is the winner? And how? As proven by Emerson and Jutla [EJ91], parity games are memoryless determined: every node has a unique winner and a corresponding memoryless winning strategy. A (memoryless) strategy for player α is a partial function σ_α from a subset of O_α to V. A play π is consistent with σ_α if for every v_i in π belonging to the domain of σ_α, v_{i+1} is $\sigma_\alpha(v_i)$. A strategy σ_α for player α is a *winning strategy* for a node v if every play in v consistent with this strategy is won by α, i.e. regardless of the moves selected by $\bar{\alpha}$. As such, a game $\mathcal{PG}$ defines a winning function $W_{\mathcal{PG}} : V \mapsto \{0, 1\}$. The set $W_{\mathcal{PG},\alpha}$ or, when $\mathcal{PG}$ is clear from the context, W_α denote the set of nodes won by α. Moreover for both players $\alpha \in \{0, 1\}$, there exists a memoryless winning strategy σ_α with domain $W_\alpha \cap O_\alpha$ that wins in all nodes won by α. A *solution* of $\mathcal{PG}$ consists of a function $W' : V \mapsto \{0, 1\}$ and two winning strategies σ_0 and σ_1, with $dom(\sigma_\alpha) = W'_\alpha \cap O_\alpha$, such that every play in $v \in W'_\alpha$ consistent with σ_α is won by α. Solutions always exist; they may differ in strategy but all have $W' = W_{\mathcal{PG}}$, the winning function of the game. We can say that the pair (σ_0, σ_1) proves that $W' = W_{\mathcal{PG}}$.

Example 1. Figure 2.1 shows a parity game with six nodes. In the game-graph we can trace a play $\langle f, e, d, c, b, a, b, c, b, a, b, c, b, a \ldots \rangle$. Three nodes, a, b and c are visited infinitely often; they have priority $3, 4$ and 1. Since the maximum of these priorities is 4, Even wins this play.
However in this play neither Odd nor Even play optimally. Odd has a winning strategy for nodes $c, \ldots, f$: the move Odd should make for c is to play to c itself. Similarly, Even should always play from b to a. As soon as Even plays to node c his opponent, Odd, can control the game and win. Figure 2.2 shows a memoryless solution for this parity game.

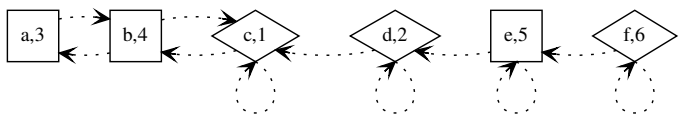


Figure 2.1: A parity game with nodes $a, \ldots, f$. Square nodes are owned by Even, diamonds by Odd. The label inside a node shows its name and priority.

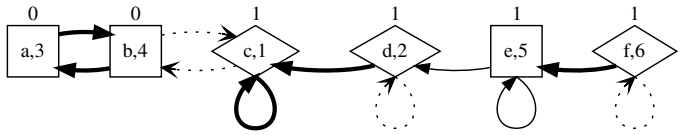


Figure 2.2: The solution of Fig. 2.1: The label on top of a node is the winner. A bold edge belongs to the winning strategy of the owner of its origin node. A slim edge starts in a node that is lost by its owner. All other edges are dotted.

2.4 Parametrised parity games

In this section, we define *parametrised parity games* (PPG) as introduced in [LBDar] in order to formalise the use of justifications (Chapter 3). Their usage is further expanded upon in Chapter 6. This is a small extension to ordinary parity games where parameters are used to override the winners of some nodes. These parameters are used to verify intermediate states of recursive parity game solving algorithms. Other applications include configuring parity games to activate and deactivate parts of the game-graph.

A PPG consists of a parity game $\mathcal{PG}$ and a parameter function P, a partial function $P : V \mapsto \{0, 1\}$ with domain $P_{01} \subseteq V$. Elements of P_{01} are called parameters, and P assigns a winner to each parameter. Plays are the same as in a $\mathcal{PG}$ except that every play that reaches a parameter v ends immediately and is won by $P(v)$.

Definition 1 (Parametrised parity game)**.** *Let $\mathcal{PG} = (V, E, O, Pr)$ be a parity game and $P : V \rightharpoonup \{0, 1\}$ a partial function with domain $P_{01} \subseteq V$. Then $(P, \mathcal{PG})$ is a* parametrised parity game*, denoted $\mathcal{PG}_P$, with parameter set P_{01}. If $P(v) = \alpha$, we call α the assigned winner of parameter v. The sets P_0 and P_1 denote parameter nodes with assigned winner 0 respectively 1.*

A play of $\mathcal{PG}_P$ is a sequence of nodes $\langle v_0, v_1, \dots \rangle$ such that for all i: if $v_i \in P_\alpha$ then the play halts and is won by α, otherwise v_{i+1} exists and $(v, v_{i+1}) \in E$. For infinite plays, the winner matches the winner in the original parity game $\mathcal{PG}$.

Every parity game $\mathcal{PG}$ defines a class of parametrised parity games, one for each partial function P. The original $\mathcal{PG}$ corresponds to one of these games, namely the one without parameters ($P_{01} = \emptyset$); every total function $P : V \mapsto \{0, 1\}$ defines a trivial PPG, with plays ending immediately and winners $W_{\mathcal{PG}_P} = P$. Of the latter, $\mathcal{PG}_{H_d}$, the PPG which assigns default winners to nodes, will be useful.

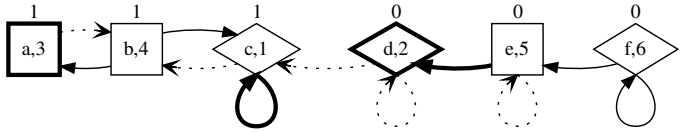


Figure 2.3: A parametrised parity game with nodes $a, \ldots, f$, $P_0 = \{d\}$ and $P_1 = \{a\}$, and winning strategies for 0 and 1. The two parameter nodes are in bold. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority; the label on top of a node is the winner. A bold edge belongs to a winning strategy (of the owner of its start node). A slim edge is one starting in a node that is lost by its owner. All remaining edges are dotted.

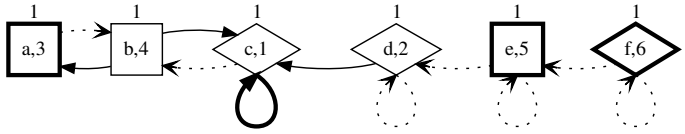


Figure 2.4: A parametrised parity game and strategy, after withdrawing d from the parameters, and adding e and f.

A PPG $\mathcal{PG}_P$ can be reduced to an equivalent parity game: in each parameter $v \in P_{01}$ replace the outgoing edges with a self-loop and the priority of v with $P(v)$. We denote this standard parity game in which P is embedded as $\mathcal{PG}^P$. Every infinite play $\langle v_0, v_1, \ldots \rangle$ in $\mathcal{PG}_P$ is also an infinite play in $\mathcal{PG}^P$ with the same winner. Every finite play $\langle v_0, v_1, \ldots, v_n \rangle$ with winner $P(v_n)$ in $\mathcal{PG}_P$ corresponds to an infinite play $\langle v_0, v_1, \ldots, v_n, v_n, \ldots \rangle$ with winner $P(v_n)$ in $\mathcal{PG}^P$. Thus, the two games are equivalent. It follows that any PPG $\mathcal{PG}_P$ is a zero-sum game defining a winning function W and having memory-less winning strategies σ_α with domain $(W_\alpha \setminus P_\alpha) \cap O_\alpha$ (for $\alpha = 0, 1$).

We introduce PPG's as the sequence of consecutive states of many parity game solving algorithms can be understood as a sequence of solved PPG's. In each step, strategies and parameters are modified and a solution for one PPG is transformed into a solution for a next PPG and this until the input PG is reached.

Example 2. Figure 2.3 shows a parametrised parity game based on the previous example and its solution. The parameter nodes a and d are won by the assigned winners, respectively 1 and 0. Player 1 owns node c and wins its priority. Hence, by playing from c to c, 1 wins in this node. Node b is owned by 0 but has only moves to nodes won by 1, hence it is also won by 1. Player 0 wins node e by playing to node d; 1 plays in node f but playing to f results in an infinite path won by 0, while playing to node e runs into a path won by 0, so f is won by 0. Assume we want to remove d from the parameters to obtain a solution for a next game. Player 1 now wins in d by playing to c. This breaks the strategies and winners of the nodes e and f that depended on d. Figure 2.4 shows one way to restore consistency by revising the parameter function P and the strategies. Here, it is done by turning e and f into parameters won by 1. There are other options, e.g., by turning f into a parameter won by 0.

2.5 Existing parity game solving algorithms

A lot of parity game solving algorithms exist. This section lists the algorithms used in later sections and other well-known and influential algorithms. After starting with fixpoint iteration in the next section, Zielonka's algorithm is discussed in Section 2.5.2, priority promotion in Section 2.5.3 and tangle learning in Section 2.5.4. Finally, in Section 2.5.5 we discuss other algorithms for solving parity games.

2.5.1 Fixpoint iteration

Fixpoint iteration is one of the simplest algorithms able to solve parity games. Even though its ability to solve parity games has been known for a long time, in its simplest form it can only determine the winners of the nodes; it is unable to determine a winning strategy. Thus it remained unimplemented until Bruse et al. [BFL14] devised a method to determine the *eventually-positional winning strategy*. Chapter 4 shows a variant of this algorithm which finds a winning strategy without an exponential amount of memory. Recently another variant by van Dijk and Rubbens [vR19] also reduced the exponential memory requirement to $O(|V| \log(|V|))$.

The algorithm consists of two insights: (i) the nodes won by player Even for a parity game can be represented as a μ-calculus formula, (ii) the nested fixpoint iteration algorithm evaluates μ-formula for finite domains in exponential time.

Walukiewicz [Wal02] characterised, for a parity game with an even highest priority d, the winning positions of Even as follows: the set of nodes won by

```
   input : A parity game
   output : S_d: nodes won by Even
 1 Fn basic((V, E, O, Pr)):
 2     for i ∈ {1 ... d} do
 3         if i is even then S_i ← V
 4         else S_i ← ∅
 5     do
 6         S_φ ← Φ(S_d, ..., S_2, S_1)
 7         i ← 1
 8         while S_i = S_φ ∧ i ≤ d do
 9             i ← i + 1
10         if i ≤ d then
11             S_i ← S_φ
12             for j ∈ {i − 1 ... 1} do
13                 if j is even then S_j ← V
14                 else S_j ← ∅
15     until i = d + 1
16     return S_d
```

Algorithm 2.1: A basic fixpoint iteration algorithm

Even is a fixpoint calculation of $\nu S_d.\mu S_{d-1}.\ \ldots \mu S_1.\Phi(S_d,\ldots,S_1)$ where Φ is a monotonic function over the lattice $(2^V,\subseteq)$:

$$\begin{aligned}\Phi(S_d,\ldots,S_1) = \quad & \{v \in O_0 | \exists w : (v,w) \in E \wedge w \in S_i \text{ with } i = Pr(w)\} \cup \\ & \{v \in O_1 | \forall w : (v,w) \in E \Rightarrow w \in S_i \text{ with } i = Pr(w)\}\end{aligned}$$

Since the inner operator Φ is monotone in all of its arguments $S_d,\ldots,S_1$, the corresponding fixpoint computation has a unique and well-defined outcome, which is also computed by Algorithm 2.1.

Bruse et al. [BFL14] observe that it suffices to store elements of priority i in S_i; indeed computing Φ only checks if $w \in S_i$ for $i = Pr(w)$, hence: $\Phi(S_d,\ldots,S_1) = \Phi(S_d^{=d},\ldots,S_1^{=1})$. Omitting the irrelevant elements, all S_i can be combined in a single set $S = \cup_i S_i^{=i}$; finally, the operation $\Phi(S_d,\ldots,S_1)$ is replaced by:

$$\begin{aligned}\phi(S) \quad = \quad & \{v \in O_0 | \exists w : (v,w) \in E \wedge w \in S\} \cup \\ & \{v \in O_1 | \forall w : (v,w) \in E \Rightarrow w \in S\}\end{aligned}$$

A second optimisation, developed by Long et al. [Lon+94], partially eliminates re-initialisations. Instead of resetting all nodes of lower priority than i to their initial state ($S_R^{<i} \leftarrow V^{<i} \cap V^{\equiv_2 0}$), only nodes with a priority of opposite parity are reset (Algorithm 2.2, Lines 6 to 9): if i is even then the nodes with even

priority are kept and nodes with an odd priority are removed; analogously, if i is odd, odd nodes are kept and nodes with an even priority are added.

The first optimisation reduces the memory use by a factor d, the second is, according to Seidl [Sei96], sufficient to reduce the time complexity to $O(|E| \cdot n^{\lceil d/2 \rceil})$ with $n = |V|/d + 1$. These two optimisations result in Algorithm 2.2, which calculates the nodes won by Even as S.

input : A parity game
output : S: the nodes won by *Even*
1 **Fn** *fixpoint_iteration*($\mathcal{PG}$)**:**
2 $S \leftarrow \{v \in V | Pr(v) \text{ is even}\}$
3 **while** $S \neq \phi(S)$ **do**
4 $S_\phi \leftarrow \phi(S)$
5 $i \leftarrow min\,\{Pr(v) | v \in S \triangle S_\phi\}$
6 **if** i is even **then**
7 $S_R \leftarrow S \setminus V^{\equiv_2 1}$
8 **else**
9 $S_R \leftarrow S \cup V^{\equiv_2 0}$
10 $S \leftarrow S^{>i} \cup S_\phi^{=i} \cup S_R^{<i}$
11 **return** S

Algorithm 2.2: The time- and memory-efficient algorithm of Bruse et al. [BFL14]. $\triangle$ is the symmetric set difference.

2.5.2 Zielonka's algorithm

Zielonka's algorithm [Zie98] determines the winners by recursively decomposing a game G in a smaller sub-game G' and first solving the sub-game. Once the winners of G' are determined, it returns to the nodes not in G' and, if needed, another sub-game is created and solved.

The algorithm is based on the notion of the *attracted node set* of a set S, denoted $Attr_\alpha(S)$. Informally, this is the set of all nodes from which α can force a play to S. Formally, this set can be computed using a least fixpoint computation:

$$\begin{aligned} Attr_\alpha(E, V, S) = \mu A.S \quad & \cup \{v \in O_\alpha \cap V | \exists w : (v, w) \in E \wedge w \in A\} \\ & \cup \{v \in O_{\bar{\alpha}} \cap V | \forall w : (v, w) \in E \Rightarrow w \in A\} \end{aligned}$$

In addition, Zielonka's algorithm computes a corresponding α-strategy $\sigma_{Attr} : (A \cap O_\alpha) \setminus S \to A$ for all attracted nodes which shows how any play starting in a node $v \in A$ eventually ends in S.

For correctness, Zielonka's algorithm depends on two properties of $Attr_\alpha$: (i) if all nodes of S can be won by α then all attracted nodes are won by α and (ii) if all nodes in S have priority $p \equiv_2 \alpha$ and all attracted nodes have a lower (or equal) priority then all moves from S to $Attr_\alpha(S)$ are won by α.

For a game G and a set of nodes A we define $G \setminus A$ as the removal of A from all parts of G: the nodes, and edges, owners, and priorities. In the original algorithm, Algorithm 2.3, empty games are immediately solved. For non-empty games, the maximal priority p is calculated. Then the algorithm calculates the set A together with a strategy σ_A for all nodes attracted to the *heads*, i.e., nodes with the maximal priority, $V^{=p}$. A play π starting in a head and consistent with σ_A has two options: either π stays in A and forms an infinite play with highest priority p, won by α, or π escapes to an unattracted node in $V \setminus A$.

input : A parity game $G = (V, E, O, Pr)$
output : W_0, W_1: winning nodes,
σ_0, σ_1: winning strategies

1 **Fn** *Zielonka*(G)**:**
2 **if** $V = \emptyset$ **then return** $(\emptyset, \emptyset, \emptyset, \emptyset)$
3 $p \leftarrow max\,\{Pr(v) | v \in V\}$
4 $\alpha \leftarrow p \bmod 2$
5 $A, \sigma_A \leftarrow Attr_\alpha(V^{<p}, V^{=p})$
6 $W_0, W_1, \sigma_0, \sigma_1 \leftarrow$ **Zielonka**$(G \setminus A)$
7 **if** $W_{\bar{\alpha}} = \emptyset$ **then**
8 $W_\alpha \leftarrow W_\alpha \cup A$
9 $F, \sigma_F \leftarrow Attr_\alpha(E, V^{=p} \cap O_\alpha, W_\alpha)$
10 $W_\alpha \leftarrow W_\alpha \cup F$
11 $\sigma_\alpha \leftarrow \sigma_\alpha \cup \sigma_A \cup \sigma_F$
12 **return** $W_0, W_1, \sigma_0, \sigma_1$
13 **else**
14 $B, \sigma_B \leftarrow Attr_{\bar{\alpha}}(E, V, W_{\bar{\alpha}})$
15 $W'_0, W'_1, \sigma'_0, \sigma'_1 \leftarrow$ **Zielonka**$(G \setminus B)$
16 $W'_{\bar{\alpha}} \leftarrow W'_{\bar{\alpha}} \cup B$
17 $\sigma'_{\bar{\alpha}} \leftarrow \sigma'_{\bar{\alpha}} \cup \sigma_B$
18 **return** $W'_0, W'_1, \sigma'_0, \sigma'_1$

Algorithm 2.3: Zielonka's algorithm

To finally determine the winner the sub-game $G \setminus A$ is recursively solved to determine W'_0 and W'_1 (Line 6). If all nodes in this sub-game are also won by α, then, there is no escape, all nodes of this game are won by α. All that remains is finding a strategy for the heads since the moves for these nodes are explicitly not included in σ_A (Line 9). In the other case, some nodes are won by the opponent: heads may be attracted or forced to $W'_{\bar{\alpha}}$ and consequently attract

more nodes of A (Line 14). These nodes are attracted to a set B and removed to create a second sub-game $G \setminus B$. The solution of this sub-game together with nodes B won by $\bar{\alpha}$ and the corresponding strategy σ_B form the final solution of G.

2.5.3 Priority promotion

Priority promotion [BDM16b; BDM16a; BDM18] is an effective algorithm for solving parity games: Based on Benerecetti et al. and the results in Chapter 5, the algorithm performs better than Zielonka's algorithm.

The algorithm achieves this by applying two ideas. First, the recursion is made implicit. This reduces the space complexity to $O(|V| \log(k))$ where k is the number of priorities. The space reduction is accompanied with a small efficiency improvement, the same technique has since been applied to Zielonka's algorithm [van18b]. Second, Benerecetti et al. introduce the notion of Quasi Dominion (QD), a relaxation of a dominion. An α-dominion is a subset of nodes for which $\bar{\alpha}$ cannot escape this set and for which an α-strategy exists that wins all nodes in the dominion. For a Quasi Dominion, this is weakened to: a quasi α-dominion is a subset of nodes for which an α-strategy exists that wins all nodes *unless* the opponent $\bar{\alpha}$ escapes this set.

In terms of recursive sub-games, in a sub-game where all quasi-dominion nodes are present but the "escape-nodes" have been removed, the quasi dominion is a regular dominion. Thus priority promotion can be described as an improvement upon Zielonka's algorithm that may return a partial solution of parity games. An outer loop (Algorithm 2.5) is responsible for finding the winners of all nodes by repeatedly solving the left-over game.

The algorithm uses an additional data-structure: the region-mapping, $r : V \to P$, is a function that determines the priority to which every node is currently attracted. The notation $r[A \mapsto p]$ is used to map the value of the nodes in the set A with p, all other nodes retain their original value.

Solving a parity game initially follows Zielonka's algorithm: nodes are attracted to the maximal priority p (Line 11). Typically, the attracted set R is neither a dominion nor a quasi-dominion. By assigning region p for all attracted nodes and selecting a new priority on Line 14 the nodes are implicitly removed. In the next iteration the smaller set of nodes SG determines a sub-game $G \cap SG$.

The process diverges when a promotable quasi-dominion R is found: within the sub-game $G \cap V'$ the winner of the nodes in R are fixed. If the QD R has no escaping nodes, then **Promote** returns the dominion R (Line 16). An

input : A parity game $G = (V, E, O, Pr)$
output : (α, R): an α-dominion R of G

1 **Fn** *Esc*(R)**:**
2 **return** $\{w | v \in O_{\bar{\alpha}}, (v, w) \in E, w \notin R\}$
3 **Fn** *promote*(G)**:**
4 $r \leftarrow Pr$
5 $p \leftarrow max(Pr(V))$
6 **while** *true* **do**
7 $\alpha \leftarrow p \mod 2$
8 $H \leftarrow \{v \in V | r(v) = p\}$
9 $V' \leftarrow \{v \in V | r(v) \leq p\}$
10 $E' \leftarrow E \cap (V' \times V')$
11 $R \leftarrow Attr_\alpha(E', V', H)$
12 $r \leftarrow r[R \mapsto p][v \mapsto Pr(v) | v \in V, r(v) < p, v \notin R]$
13 **if** $\exists w \in Esc(R) : r(w) < p$ **then**
14 $p \leftarrow max\{r(v) | r(v) < p\}$
15 **else if** $Esc(R) = \emptyset$ **then**
16 **return** (α, R) `// if` R `is an` α`-dominion`
17 **else** `// if` R `is a promotable quasi-dominion`
18 $p \leftarrow min(r(Esc(R)))$
19 $r \leftarrow r[R \mapsto p]$

Algorithm 2.4: Priority promotion

input : A parity game $G = (V, E, O, Pr)$
output : W_0, W_1, a partition of V according to winner Even and Odd

1 **Fn** *CompletePP*(G)**:**
2 $W_0 \leftarrow \emptyset$, $W_1 \leftarrow \emptyset$
3 **while** $G \neq \emptyset$ **do**
4 $G' \leftarrow G \setminus (W_0 \cup W_1)$
5 $(\alpha, D) \leftarrow promote(G')$
6 $W_\alpha \leftarrow W_\alpha \cup D$
7 $W_\alpha \leftarrow Attr_\alpha(E, V, W_\alpha)$
8 **return** W_0, W_1

Algorithm 2.5: Full priority promotion solver without winning strategies

external loop is responsible for correctly removing R from G in order to further solve the game. Else, the promotion priority of the QD R is calculated and R is then merged (Lines 18 and 19). Progress and termination are guaranteed by defining a finite state space (r, p) and partial order and proving the state increases monotonically in Algorithm 2.4.

Similarly to Zielonka's algorithm, keeping track, for each node, of the last move used to attract the node results in a winning strategy. This uses at most $O(|V|\log(|V|))$ additional bits of memory.

2.5.4 Tangle learning

Tangle learning [van18a; van18b] is a recent state-of-the-art algorithm based on deriving and attracting tangles. Tangles are sub-games won by a single player α with a strongly connected strategy:

Definition 2 (Tangles)**.** *An α-tangle is a set of nodes $\tau \subseteq V$ for which there exists a tangle-strategy $\sigma : \tau|_\alpha \to \tau$ such that the graph (τ, E_τ) with $E_\tau = E \cap (\sigma \cup ((\tau \cap O_{\bar{\alpha}}) \times \tau))$ has at least one edge, is strongly connected, and all cycles within (τ, E_τ) are won by α.*

A play starting in a node of an α-tangle can either stay within the tangle or $\bar{\alpha}$ can escape to a node in $Esc(\tau)$. Thus if all escapes are won by α then all nodes in the α-tangle are won by α; if a tangle has no escapes, it is a dominion. As every tangle is a quasi-dominion, the promotion region of a tangle t is the minimal region of all escaping nodes, $min(r(Esc(t)))$.

Integrating the tangles involves an extension to the attraction function used by Zielonka's algorithm. Recall that nodes are attracted for a player α if that player can force a play to a given set, rules (i) and (ii). For tangle learning the property is extended to either playing to the given set or α wins the infinite play. This property is satisfied for α-tangles: all nodes of an α-tangle are attracted if all escapes of the tangle are attracted, rule (iii).

Formally, given the set of tangles T (with the α-tangles of T denoted as $T|_\alpha$), the set of nodes V, the set of edges E, the subset of nodes attracted to S is:

$$
\begin{aligned}
&TAttr^T_\alpha\,(E, V, S) = \mu A.\ S \\
&\qquad \cup \{v \in V \cap O_\alpha \mid \exists w : (v, w) \in E \wedge w \in A\} && (i)\\
&\qquad \cup \{v \in V \cap O_{\bar{\alpha}} \mid \forall w : (v, w) \in E \Rightarrow w \in A\} && (ii)\\
&\qquad \cup \{v \in V \cap t \mid t \in T|_\alpha \wedge (Esc(t) \cap V) \neq \emptyset \wedge (Esc(t) \cap V) \subseteq A\} && (iii)
\end{aligned}
$$

Similar to *Attr*, *TAttr* records how nodes are attracted as a strategy σ_A. The strategy for attracted nodes is unchanged while the strategy for attracted nodes of a tangle is determined by σ_t.

The second part of the algorithm consists of learning tangles. They are found after attracting: assume $A, \sigma_A = TAttr^T_\alpha(E, V, S)$. If a node $v \in S_\alpha$ can play

within A or if a node $v \in S_{\bar{\alpha}}$ must play within A then the possibility of a cycle exists. Furthermore, if all nodes in S have the same priority $p \equiv_2 \alpha$ and all attracted nodes A have a lower priority then all cycles are won by α. The tangle learning algorithm will only attract nodes and tangles of a lower priority by invariant. Thus every non-trivial cycle will be an α-tangle. The problem then consists of determining the non-trivial SCC's in (A, E_A) with $E_A = E \cap (\sigma_A \cup (A_{\bar{\alpha}} \times A))$. The strategy σ_A also determines a strategy σ_t for every extracted tangle t.

input : A parity game
$\mathrm{G} = (V, E, V_0, V_1, Pr)$
output : (α, Dom, T)
An α-dominion Dom of G
An extended set of tangles T in G

Fn *tangle_learning*(G, T)**:**
  $r \leftarrow Pr, p \leftarrow max\,\{Pr(v) | v \in V\}$
  while *true* **do**
    $\alpha \leftarrow p \bmod 2$
    $H \leftarrow \{v \in V | r(v) = p\}$
    $V' \leftarrow \{v \in V | r(v) \leq p\}$
    $E' \leftarrow E \cap (V' \times V')$
    $A, \sigma_A \leftarrow TAttr^T_\alpha(E', V', H)$
    $New \leftarrow extract_tangles(A, \sigma_A)$
    $T \leftarrow T \cup New$
    if $New = \emptyset$ **then**
      $r \leftarrow r[A \mapsto p][v \mapsto Pr(v) | v \in V, r(v) < p, v \notin R]$
      $p = max\,\{Pr(v) | v \in V, r(v) < p\}$
    else if $\exists t \in New : Esc(t) = \emptyset$ **then**
      $Dom \leftarrow \{(v \in t | t \in New, Esc(t) = \emptyset\}$
      return (α, Dom, T)
    else `// next iteration, new tangles are attracted`
      $p \leftarrow max\,\{min(r(Esc(t))) | t \in New\}$

Algorithm 2.6: On-the-fly (partial) tangle learning

Tangle learning consists of first finding tangles and then, to progress, attracting them. If a tangle is used a second time, some work is saved. The composite tangles (containing sub-tangles) save more work. Algorithm 2.6 shows a tangle learning variant similar to Algorithm 2.4. First, we explain some key variables: (i) the set T contains tangles found so far, (ii) similar to priority promotion, the function r contains the region of each node. Again, the algorithm solves for a partial solution; when a dominion is found, it is returned immediately. An outer loop (Algorithm 2.7) is responsible for accumulating all dominions and shrinking the remaining parity game.

```
   input : A parity game
           G = (V, E, O, Pr)
1  Fn CompleteTL(G):
2  |  W_0←∅, W_1←∅
3  |  T←∅
4  |  while G ≠ ∅ do
5  |  |  G'←G \ (W_0 ∪ W_1)
6  |  |  (α, D, T)←tangle_learning(G', T)
7  |  |  W_α←W_α ∪ D
8  |  |  W_α←Attr_α(E, V, W_α)
9  |  return W_0, W_1
```

Algorithm 2.7: Tangle learning solver without winning strategies

A regular iteration of the algorithm runs from Lines 3 to 13: First, all nodes from lower regions are attracted to region p. Next, Line 27, searches for promotable tangles in the current region. When no tangles are extracted, the nodes are assigned region p and the next region is selected. This induces the invariant that no nodes can be attracted to higher regions than p. However, in case tangles are found, the set of tangles T grows. All dominions in the new tangles are returned. Otherwise, the next region is selected to maintain the previously mentioned invariant. This is the highest promotion region of all tangles in New.

Note how T contains all tangles extracted during the execution of the algorithm. An effective optimisation is clearing useless tangles from this set. According to van Dijk [van18b], tangles with nodes from W_0 and W_1 should be removed. Or, as games become more difficult, old unused tangles could be removed similar to aggressive clause deletion techniques from SAT-solving [AS09]. However it seems most applications do not involve difficult games (See Michaud and Colange [MC18] and Chapter 5).

2.5.5 Other algorithms

Progress measures [Jur00; JL17] are another family of algorithms for solving parity games. Moreover, recently, a quasi-polynomial progress measure was found by Fearnley et al. [Fea+19]. Multiple variants are implemented in Oink [van18b] and PGSolver [FL09a].

These algorithms assign an evaluation, *a progress measure*, to every node. This progress measure encodes an underestimation of the best advantageous path that can be forced by one player. Progress measures are updated monotonically according to a crafted bounded lattice. Every update is evidence that a longer

path has been found. When the process reaches fixpoint, all nodes won by the main player have reached the greatest progress measure. The residual nodes are won by the opponent. A winning strategy can be extracted from the progress measure of the residual nodes.

Strategy iteration (or strategy improvement) is originally an algorithm by Hoffman and Karp [HK66] for solving stochastic games. This algorithm was adapted by Vöge and Jurdziński [VJ00] for parity games. The following projects implement this algorithm: Oink [van18b], PGSolver [FL09a] and Strix [MSL18].

Strategy improvement works by starting with an initial random strategy for one player. After which this strategy is evaluated and a node is sought for which changing the strategy improves the evaluation. This is iterated until reaching a fixpoint. When the strategy is optimal, the evaluation determines which nodes are won by the player. It is also possible to determine a winning strategy for the opponent from the residue.

Parys' algorithm [Par19] is a quasi-polynomial variant of Zielonka's algorithm. The main observation concerns the sizes of the dominions: Parys observed that a dominion of size n has at most one strongly connected component of size larger than $n/2$; all others must be smaller than $n/2$. This idea is incorporated in an algorithm that finds dominions limited in size.

Intuitively, finding all dominions of size at most n works as follows: like Zielonka's algorithm, initially all nodes of the highest priority p, and their attracted nodes are assumed as won by the winner of priority p. The remaining nodes form a sub-game to be solved recursively. The algorithm determines for which nodes this assumption is false in three phases: the first phase finds all dominions of the sub-game of size less than $n/2$, every invalidated node updates the sub-game. This is repeated until no more nodes with invalid assumptions are found. The second phase searches all dominions of size less than n once. Since the invalidated assumptions may unlock more small dominions, sub-game is again updated and the first phase is then repeated. All dominions of size less than $n/2$ are sought until no more invalid assumptions are found. The remaining assumptions are then either part of dominions of size less than n won by the winner for priority p, or part of dominions larger than n for either player. Each dominion with less than n then consist of strongly connected components (sub-dominions) of which all but one have size less than $n/2$. Parys proved in detail that this idea leads to a quasi-polynomial algorithm that solves parity games.

An implementation of this algorithm can be found in Oink [van18b].

Chapter 3

Solving parity games with justifications

The contents up to Section 3.4.2 make up the first part of Lapauw et al. [LBDar]. Sections 3.5 and 3.6 broach two new topics: the first proves a well-known optimisation used extensively when implementing the algorithms of Chapter 4. The second shows how, in the future, the framework of this chapter can be used to improve model synthesis tools.

3.1 Introduction

The idea of justifications originates from Denecker and De Schreye [DD93]. This paper defined a structure which describes the derivation for the truth of a fact in a logic program. Later, applications condensed this derivation to a graph-structure: it has been applied to find satisfiable models for inductive definitions, SAT(ID), by Mariën et al. [Mar+08], to find models for nested fixpoint definitions by Hou et al. [HDD10]... A justification graph justifies the interpretation of a logic program: in a correct justification the graph unfolds step by step why a fact is true or false. Furthermore different global properties (thus far constraints on cycles) of a justification correspond to different popular semantics for logic programs (such as the well-founded, stable or completion semantics).

This chapter shows how, due to the close relationship between nested fixpoints and parity games, justifications can be adapted for parity games. In essence, a

justification is a flexible representation for solving parity games: a justification pairs the winner function with a winning strategy and correct justifications are isomorphic to a solution for a parametrised parity game where the parameter function is also determined by the justification. Furthermore, to aid solving a parity game, the representation is easily modified. We show one operation that removes one node as a parameter, sometimes at the cost of re-parametrising other nodes. With additional requirements repeating this operation eventually solves the parity game. This flexibility allows using a justification as the core of a number of algorithms, as shown in the next chapter.

Section 3.2 formalises the concept of a justification as a solution for a (parametrised) parity game. Next, Section 3.3 and 3.4 show how to manipulate this justification while keeping the necessary invariants intact. Finally, we include two related topics on justifications: first, a practical optimisation to be used when implementing algorithms based on justifications and, second, the idea of solving a modified parity game without restarting. We conclude in Section 3.7.

3.2 Justifications

In Figure 3.1, the label on top of each node denotes the winner, the solid edges form the subgraph that confirms the winner of these nodes. We formalise the winners and the subgraph as a *justification*. In the rest of the chapter, we assume the existence of a parity game $\mathcal{PG} = (V, E, O, Pr)$ and a parametrised parity game $\mathcal{PG}_P = (\mathcal{PG}, P)$ with P a parameter function with set of parameters P_{01}. Also, we use $H : V \mapsto \{0, 1\}$ as a function describing a "hypothesis" of who is winning in the nodes.

Definition 3 (Direct justification)**.** *A* direct justification *dj for player α to win node v is a set containing one outgoing edge of v if $O(v) = \alpha$ and all outgoing edges of v if $O(v) = \bar{\alpha}$.*

A direct justification dj wins v for α under hypothesis H *if for all $(v, w) \in dj$, $H(w) = \alpha$. We also say:* α wins v by dj under H.

Definition 4 (Justification)**.** *A* justification *J for $\mathcal{PG}$ is a tuple (V, D, H) such that (V, D) is a subgraph of (V, E). If a node has outgoing edges in D, it is* justified *in J, otherwise it is* unjustified.

Definition 5 (Weakly winning)**.** *A justification $J = (V, D, H)$ is* weakly winning *if for all justified nodes $v \in V$ the set of outgoing edges $Out_J(v)$ is a direct justification that wins v for $H(v)$ under H.*

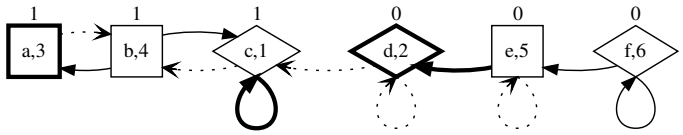


Figure 3.1: A parametrised parity game with nodes $a, \ldots, f$, $P_0 = \{d\}$ and $P_1 = \{a\}$, and winning strategies for 0 and 1. The two parameter nodes are in bold. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority; the label on top of a node is the winner. A bold edge belongs to a winning strategy (of the owner of its start node). A slim edge is one starting in a node that is lost by its owner. All remaining edges are dotted.

We observe that any justification $J = (V, D, H)$ determines a PPG $\mathcal{PG}_{P_J}$ where the parameter function P_J is the restriction of H to unjustified nodes.

If J is weakly winning, the set of edges $\{(v, w) \in D \mid O(v) = H(v) = \alpha\}$ is a partial function on O_α, i.e., a strategy for α. We denote it as $\sigma_{J,\alpha}$.

Proposition 3.1. *Assume a weakly winning justification $J = (V, D, H)$. Then, (i) For every path π in D, all nodes v on π have the same hypothetical winner $H(v)$. (ii) All finite paths π starting in node v in D are won in $\mathcal{PG}_{P_J}$ by $H(v)$. (iii) Every path in D with nodes hypothetically won by α is consistent with $\sigma_{J,\alpha}$. (iv) Every play starting in v of $\mathcal{PG}_{P_J}$ consistent with $\sigma_{J,H(v)}$ is a path in D.*

Proof. (i) Since any edge $(v, w) \in D$ belongs to a direct justification that wins v for $H(v)$, it holds that $H(v) = H(w)$. It follows that every path π in D consists of nodes with the same hypothetical winner. (ii) If path π in v is finite and ends in parameter w, then $H(v) = H(w)$. The winner of π in $\mathcal{PG}_{P_J}$ is $P_J(w)$ which is equal to $H(v)$ as H expands P_J. (iii) Every path in D with hypothetical winner α, follows $\sigma_{J,\alpha}$ when it is in a node v with owner α. (iv) Let $H(v) = \alpha$ and π be a play in v of $\mathcal{PG}_P$ consistent with $\sigma_{J,\alpha}$. We can inductively construct a path from $v = v_1$ in D. It follows from (i) that the n'th node v_n has $H(v_n) = H(v_1) = \alpha$. For each non-parameter node v_n, if $O(v_n) = \alpha$, then $v_{i+1} = \sigma_{J,\alpha}(v_i)$ which is in D. If $O(v_n) = \bar{\alpha}$ then D contains all outgoing edges from v_n including the one to v_{n+1}. □

Definition 6 (Winning)**.** *A justification $J = (V, D, H)$ is* winning *if (i) J is weakly winning and (ii) all infinite paths $\langle v_1, v_2, \ldots \rangle$ in D are plays of $\mathcal{PG}$ won by $H(v_1)$.*

Observe that, if J is winning and $H(v) = \alpha$, all plays in $\mathcal{PG}_{P_J}$ starting in v and consistent with $\sigma_{(V,D,H),\alpha}$ are paths in (V, D) won by α. Hence:

Theorem 3.2. *If $J = (V, D, H)$ is a winning justification for $\mathcal{PG}_{P_J}$ then H is $W_{\mathcal{PG}_{P_J}}$, the winning function of $\mathcal{PG}_{P_J}$, with corresponding winning strategies $\sigma_{J,0}$ and $\sigma_{J,1}$.*

The central invariant of the algorithm presented below is that its data structure $J = (V, D, H)$ is a winning justification. Thus, in every stage, H is the winning function of $\mathcal{PG}_{P_J}$ and the graph (V, D) comprises winning strategies $\sigma_{J,\alpha}$ for both players. In a sense, (V, D) provides a proof that H is $W_{\mathcal{PG}_{P_J}}$.

3.3 Operations on weakly winning justifications

We introduce an operation that modifies a justification $J = (V, D, H)$ and hence also the underlying game $\mathcal{PG}_{P_J}$. Let v be a node in V, α a player and dj either the empty set or a direct justification. We define $J[v : dj, \alpha]$ as the justification $J' = (V, D', H')$ where D' is obtained from D by replacing the outgoing edges of v by the edges in dj, and H' is the function obtained from H by setting $H'(v) := \alpha$. Modifications for a set of nodes are independent of application order. E.g., $J[v : \emptyset, H_d(v) \mid v \in S]$ removes all out-going edges of v and sets $H'(v) := H_d(v)$, the default winner, for all $v \in S$. Multiple operations, like $J[v : dj, \alpha][v' : dj', \alpha']$, are applied left to right. Some useful instances, with their properties, are below.

In the proposition, a cycle in J is a finite sequence of nodes following edges in J that ends in its starting node.

Proposition 3.3. *Let J be a weakly winning justification.*

(i) If $H(v) = \bar{\alpha}$, v has no incoming edges and dj wins v for α under H, then $J[v : dj, \alpha]$ is weakly winning and there are no cycles in J' with edges of dj.

(ii) Let S be a set of nodes closed under incoming edges (if $v \in S$ and $(w, v) \in D$, then $w \in S$). It holds that $J[v : \emptyset, H_d(v) \mid v \in S]$ is weakly winning. There are no cycles in J' through edges of dj.

(iii) If $H(v) = \alpha$ and dj wins v for α under H, then $J[v : dj, \alpha]$ is weakly winning. There are no new cycles if $(v, v) \notin dj$ and no $w \in dj$ can reach v in J. Otherwise new cycles pass through v and have at least one edge in dj.

Proof. We exploit the fact that J and J' are very similar.

(i) The direct justification dj cannot have an edge ending in v since $H(v) \neq H(w)$ for $(v, w) \in dj$ and no $w \in dj$ can reach v in J since v has no incoming edges, hence J' has no cycles through dj. As J is weakly winning and H is updated only in v, the direct justification of a justified node $w \neq v$ in J is still winning in J'. Since also dj wins v for α, J' is weakly winning.

(ii) Setting $H(v)$ arbitrary cannot endanger the weak support of J' as v has no direct justification and no incoming edges in J'. Hence J' is weakly winning. Also, removing direct justifications cannot introduce new cycles.

(iii) Let $H(v) = \alpha$ and dj wins v for α under H. Let $J' = J[v : dj, \alpha]$. We have $H' = H$ so the direct justifications of all nodes $w \neq v$ in J' win w for $H'(w)$. Since dj wins v for $H'(v)$, J' is weakly winning. Also, new cycles if any, pass through dj and v.

□

3.4 Constructing winning justifications

The eventual goal of a justification is to create a winning justification without unjustified nodes. Such a justification contains a solution for the parity game without parameters. To reach this goal we start with an empty winning justification and iteratively assign a direct justification to one of the nodes.

However, haphazardly (re)assigning direct justifications will violate the intended winning justification invariant. Three problems appear: first, changing the hypothesis of a node may violate weakly winning for incoming edges. The easiest fix is to remove the direct justification of nodes with edges to this node. Yet removing direct justifications decreases the justification progress. Thus a second problem is ensuring progress and termination despite these removals. Third, newly created cycles must be winning for the hypothesis. To solve these problems, we introduce safe justifications; we start with some auxiliary concepts.

Let J be a justification. The set of nodes *reaching* v in J, including v, is closed under incoming edges and is denoted with $J{\downarrow_v}$. The set of nodes *reachable* from v in J, including v, is denoted with $J{\uparrow_v}$. We define $Par_J(v)$ as the parameters reachable from the node v, formally $Par_J(v) = J{\uparrow_v} \cap P_{01}$. The *justification level* $jl_J(v)$ of a node v is the lowest priority of all its parameters and $+\infty$ if v has none. The *justification level* $jl_J(dj)$ of a direct justification $dj = \{(v, w_1), \ldots, (v, w_n)\}$ is $min\{jl_J(w_1), \ldots, jl_J(w_n)\}$, the minimum of the justification levels of the w_i. We drop the subscript J when it is clear from the context and write $Par(v)$, $jl(v)$ and $jl(dj)$ for the above concepts. The *default*

winner of a node v is the winner of its priority, i.e., $Pr(v)$ mod 2; the *default hypothesis* H_d assigns default winners to all nodes, i.e., $H_d(v) = Pr(v)$ mod 2.

Definition 7 (Safe justification)**.** *A justification is safe iff (i) it is a winning justification, (ii) all unjustified nodes v have $H(v) = H_d(v)$, that is, the winners of the current parameters of the PPG are their default winners, and (iii) $\forall v \in V : jl(v) \geq Pr(v)$, i.e., the justification level of a node is at least its priority.*

Fixing the invariants is easier for safe justifications. Indeed, for nodes w on a path to a parameter v, $Pr(v) \geq jl(w) \geq Pr(w)$, so when v is given a direct justification to w then $Pr(v)$ is the highest priority in the created cycle and $H(v)$ correctly denotes its winner. Furthermore, the empty safe justification $(V, \emptyset, H_d)$ will serve as initialisation of the solving process.

3.4.1 The operation Justify

To progress towards a solution, we introduce a single operation, namely **Justify**. Given appropriate inputs, it can assign a direct justification to an unjustified node or replace the direct justification of a justified node. Furthermore, if needed, it manipulates the justification in order to restore its safety.

Definition 8 (**Justify**)**.** *The operation* **Justify**(J, v, dj) *is* executable *if*

- *Precondition 1: $J = (V, D, H)$ is a safe justification, v is a node in V, there exists a player α who wins v by dj under H.*
- *Precondition 2: either v is unjustified in J and $jl(dj) = jl(v) = Pr(v)$, or $jl(dj) > jl(v)$.*

Let **Justify**(J, v, dj) *be executable. If $H(v) = \alpha$ then* **Justify**$(J, v, dj) = J[v : dj, \alpha]$*, i.e., dj becomes the direct justification of v.*

If $H(v) = \bar{\alpha}$, then **Justify**$(J, v, dj) = J[w : \emptyset, H_d(w) \mid w \in J{\downarrow_v}][v : dj, \alpha]$*, i.e., α wins v by dj, while all other nodes w that can reach v become unjustified, and their hypothetical winner $H(w)$ is reset to their default winner.*

If **Justify**(J, v, dj) is executable, we say that v is *justifiable with dj* or *justifiable* for short; when performing the operation, we *justify* v.

Observe, when **Justify** modifies the hypothetical winner $H(v)$, then, to preserve weak winning, edges (w, v) need to be removed, which is achieved by removing the direct justification of w. Moreover, to preserve (iii) of safety, this process

must be iterated until fixpoint and to preserve (ii) of safety, the hypothetical winner $H(w)$ of w needs to be reset to its default winner. This produces a situation satisfying all invariants. Furthermore, when **Justify** is applied on a justified v, it preserves $H(v)$ but it replaces v's direct justification by one with a strictly higher justification level. As the proof below shows, this ensures that no new cycles are created through v so we can guarantee that all remaining cycles still have the correct winner. So, cycles can only be created by justifying an unjustified node.

Lemma 3.4. *An executable operation* $\mathbf{Justify}(J, v, dj)$ *returns a safe justification.*

Proof. Assume $\mathbf{Justify}(J, v, dj)$ is executable, $J' = \mathbf{Justify}(J, v, dj)$ and let α be the player that wins v by dj. First, we prove that J' is also a winning justification, i.e., that J' is weakly winning and that the winner of every infinite path in J' is the hypothetical winner $H(w)$ of the nodes w on the path.

The operations applied to obtain J' are the ones that have been analysed in Proposition 3.3 and for which it was proven that they preserve weakly winning. Note that, in case $H(v) = \bar{\alpha}$, the intermediate justification $J[v : \emptyset, H_d(v) \mid v \in J{\downarrow_v}]$ removes all incoming edges of v. Hence, J' is weakly winning and all nodes v, w connected in J have $H'(v) = H'(w)$ (*). If no edge in dj belongs to a cycle, then every infinite path $\langle v_1, v_2, \dots \rangle$ in J' has an infinite tail in J starting in $w \neq v$ which is, since J is winning, won by $H(w)$. By (*), this path is won by $H(v_1) = H(w)$ and J' is winning.

If J' has cycles through edges in dj, then, according to Proposition 3.3, we are in the case $H(v) = \alpha$. We analyse the nodes n on a such cycle. By safety of J, $Pr(n) \leq jl_J(n)$; as n reaches v in J, $jl_J(n) \leq jl_J(v)$. If v is unjustified in J then $jl_J(v) = Pr(v) \geq Pr(n)$, hence $Pr(v)$ is the highest priority on the cycle and, by precondition 2, $H(v)$ wins the cycle. If v is justified in J and $(v, w) \in dj$ is on the new cycle, then $jl_J(w) \geq jl_J(dj) > jl_J(v)$ (Precondition 2 of **Justify**). But w reaches v so $jl_J(w) \leq jl_J(v)$, which is a contradiction.

Next, we prove that J' is a safe justification (Definition 7). (i) We just proved that J' is a winning justification. (ii) For all unjustified nodes v of J', it holds that $H'(v) = H_d(v)$, its default winner. Indeed, J has this property and whenever the direct justification of a node w is removed, $H'(w)$ is set to $H_d(w)$.

(iii) We need to prove that for all nodes w, it holds that $jl_{J'}(w) \geq Pr(w)$. We distinguish between the two cases of $\mathbf{Justify}(J, v, dj)$.

(a) Assume $H(v) = \alpha = H'(v)$ and $J' = J[v : dj, H(v)]$ and let w be an arbitrary node of V. If w cannot reach v in J', the parameters that w reaches in J and J' are the same and it follows that $jl_{J'}(w) = jl_J(w) \geq Pr(w)$. So, (iii) holds for

w. Otherwise, if w reaches v in J', then w reaches v in J and any parameter x that w reaches in J' is a parameter that w reaches in J or one that an element of dj reaches in J. It follows that $jl_{J'}(w)$ is at least the minimum of $jl_J(w)$ and $jl_J(dj)$. As w reaches v in J, $jl_J(w) \leq jl_J(v)$. Also, by Precondition 2 of **Justify**, $jl_J(v) \leq jl_J(dj)$. It follows that $jl_{J'}(w) \geq jl_J(w) \geq Pr(w)$. Thus, (iii) holds for w.

(b) Assume $H'(v) \neq H(v) = \bar{\alpha}$ and $J' = J[w : \emptyset, H_d(w) \mid w \in J{\downarrow_v}][v : dj, \alpha]$ then for nodes w that cannot reach v in J, $Par_{J'}(w) = Par_J(w)$ hence $jl_{J'}(w) = jl_J(w) \geq Pr(w)$ and (iii) holds for w. All nodes $w \neq v$ that can reach v in J are reset, hence $jl_{J'}(w) = Pr(w)$ and (iii) holds. As for v, by construction $jl_{J'}(v) = jl_J(dj) \geq jl_J(v)$; also $jl_J(v) \geq Pr(v)$ hence (iii) also holds. □

3.4.2 Using Justify to solve parity games

In order to show **Justify** solves parity games, two additional properties are proved: i) unless the parity game is solved, Lemma 3.5 proves there exists a next step, and ii) Lemma 3.6 proves each step increases w.r.t. a finite order. Theorem 3.7 uses both properties to prove that eventually a winning justification without parameters is found.

Lemma 3.5. *Let J be a safe justification for a parametrised parity game. Unless J defines the parametrised parity game $PG_\emptyset = \mathcal{PG}$, there exists a node v justifiable with a direct justification dj, i.e., such that* **Justify**(J, v, dj) *is executable.*

Proof. If J defines the parametrised parity game $PG_\emptyset$ then all nodes are justified and J is a solution for the original $\mathcal{PG}$. Otherwise let p be the minimal priority of all unjustified nodes, and v an arbitrary unjustified node of priority p and let its owner be α. Then either v has an outgoing edge (v, w) to a node w with $H(w) = \alpha$, thus a winning direct justification for α, or all outgoing edges are to nodes w for which $H(w) = \bar{\alpha}$, thus v has a winning direct justification for $\bar{\alpha}$. In both cases, this direct justification dj has a justification level larger or equal to p since no parameter with a smaller priority exist. So **Justify**(J, v, dj) is executable. □

While the unjustified nodes with the lowest priority are always justifiable, other nodes may also be justifiable. The order in which nodes are justified may greatly affect the number of steps to complete the justification.

To show progress and termination, we need an order over justifications.

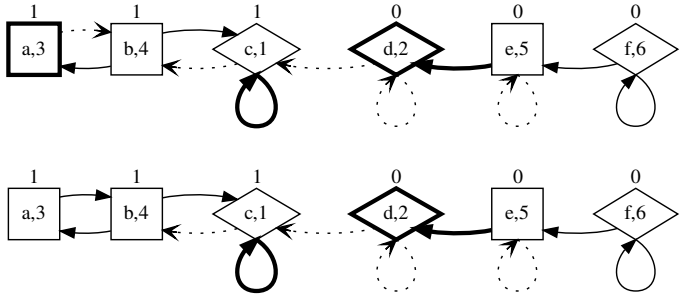


Figure 3.2: Above, in solid line the edges of the justification graph of the winning but unsafe justification of Figure 3.1 and below the result of justifying node a, a non-winning justification! The play $a, b, a, b, \ldots$ is won by Even while $H(a) = H(b) = 1$.

Definition 9 (Justification size and order over justifications)**.** *Let $1, \ldots, d$ be the range of the priority function of a parity game PG ($+\infty > d$) and J a winning justification for a parametrised parity game extending PG. The size of J, $s(J)$ is the tuple $(s_{+\infty}(J), s_d(J), \ldots s_1(J))$ where for $i \in \{1, \ldots, d, +\infty\}$, $s_i(J)$ is the number of justified nodes with justification level i.*

The order over justifications is the lexicographic order over their size: with i the highest index such that $s_i(J) \neq s_i(J')$, we have $J >_s J'$ iff $s_i(J) > s_i(J')$.

The order over justification sizes is a total order. Also, it holds that $\Sigma_i s_i(J) \leq \#V$, so the number of different sizes is bounded; the minimal value is $(0, \ldots, 0)$ and the maximal value is $(\#V, 0, \ldots, 0)$.

Example 3. Let us revisit Example 2. The winning justification J of Figure 3.1 is shown at the top of Figure 3.2. For the justified nodes of J, we have $jl(b) = 3$, $jl(c) = +\infty$, $jl(e) = 2$ and $jl(f) = 2$, so $s(J) = (1, 0, 0, 0, 1, 2, 0)$. The justification is not safe as, e.g., $jl(b) = 3 < Pr(b) = 4$. Both unjustified nodes a and d have a winning direct justification, the direct justification $\{(a, b)\}$ wins a for player 1 and the direct justification $\{(d, c)\}$ wins d for 1. The figure at the bottom shows the justification resulting from inserting the direct justification winning a. There is now an infinite path $\langle a, b, a, b, \ldots \rangle$ won by Even but with nodes with hypothetical winner Odd. This shows that condition (iii) of safety of J is a necessary precondition for performing **Justify**(J, v, dj).

Lemma 3.6. *Let J be a safe justification with size s_J, v a node justifiable with dj and $J' =$ **Justify**(J, v, dj) a justification with size $s_{J'}$. Then $s_{J'} > s_J$.*

Proof. In case v is unjustified in J and is assigned a dj that wins v for $H(v)$, v is not counted for the size of J but is counted for the size of J'. Moreover, other nodes keep their justification level (if they cannot reach v in J) or may increase their justification level (if they can reach v in J). In any case, $s_{J'} > s_J$.

In case v is justified in J and is assigned a dj that wins v for $H(v)$, then $jl_J(dj) > jl_J(v)$, so $jl_{J'}(v) > jl_J(v)$. Other nodes keep their justification level or, if they reach v, may increase their justification level. Again, $s_{J'} > s_J$.

Finally, the case where dj wins v for the opponent of $H(v)$. Nodes can be reset; these nodes w have $jl_J(w) \leq Pr(v)$. As a node cannot have a winning direct justification for both players, v is unjustified in J. Hence, by precondition (2) of **Justify**, $jl_J(dj) \geq Pr(v)$. In fact, it holds that $jl_J(dj) > Pr(v)$. Indeed, if some $w \in dj$ would have a path to a parameter of v's priority, that path would be won by $H_d(v) = H(v)$ while $H(w)$ is its opponent. Thus, the highest index i where s_i changes is $jl_J(dj)$, and s_i increases. Hence, $s_{J'} > s_J$. □

Theorem 3.7. *Any iteration of* **Justify** *steps from a safe justification, in particular from $(V, \emptyset, H_d)$, with H_d the default hypothesis, eventually solves $\mathcal{PG}$.*

Proof. By induction: Let $\mathcal{PG} = (V, E, O, Pr)$ be a parity game. Clearly, the empty justification $J^0 = (V, \emptyset, H_d)$ is a safe justification. This is the base case.

Induction step: Let J^i be the safe justification after i successful **Justify** steps and assume that $J^i = (V, D^i, H^i)$ contains an unjustified node. By Lemma 3.5, there exists a pair v and dj such that v is justifiable with dj. For *any* pair v and dj such that $\textbf{Justify}(J^i, v, dj)$ is executable, let $J^{i+1} = \textbf{Justify}(J^i, v, dj)$. By Lemma 3.4, J^{i+1} is a safe justification. By Lemma 3.6, there is a strict increase in size, i.e., $s(J^{i+1}) > s(J^i)$.

Since the number of different sizes is bounded, this eventually produces a safe $J^k = (V, D^k, H^k)$ without unjustified nodes. The parametrised parity game $\mathcal{PG}_{P_{J^k}}$ determined by J^k is $\mathcal{PG}$. Hence, H^k is the winning function of $\mathcal{PG}$, and J^k comprises winning strategies for both players. □

Theorem 3.7 gives a basic algorithm to solve parity games. The algorithm has three features: it is (1) simple, (2) *nondeterministic*, and (3) in successive steps it may arbitrarily switch between different priority levels. Hence, by imposing different strategies, different instantiations of the algorithm are obtained.

In the next chapter we simulate existing recursive algorithms by different strategies for selecting nodes to be justified. However, one difference between the simulated algorithms and their originals is in computing the set R of nodes

that are reset when dj wins v for the opponent of $H(v)$. The largest reset set for which the proofs in this paper remain valid is $\{w \in V \mid jl(w) < jl(dj)\}$. Some of the original algorithms reset even more nodes. To the best of our knowledge, the only algorithms that reset as few nodes as $\mathbf{Justify}(J, v, dj)$ are the ones we presented in [LBD20]. As the experiments presented there and in Chapter 5 show, the work saved across iterations by using justifications results in better performance.

3.5 Justifying multiple nodes at once

When implementing algorithms, optimisations are important. In this section we prove with two lemma's the possibility to reorder the assignments of the justification graph. These lemma's are then applied in the benchmarked implementations discussed in Chapter 5.

First, **Justify** is restated as a unified action consisting of an assignment followed by resetting nodes.

Lemma 3.8. Justify *can be implemented to assign the direct justification and hypothesis before resetting the dependent nodes in the resulting justification graph:*

$$\mathbf{Justify}(J, v, dj) = J[r : \emptyset, H_d(r) \mid r \in J{\downarrow_v}, H(r) = \bar{\alpha}][v : dj, \alpha]$$

Proof. The proof consists of two cases: Firstly, if $H(v) = \alpha$ then no nodes are reset. Indeed, while nodes may reach v ($J{\downarrow_v}$ may be non-empty), all these nodes have hypothesis α. Thus $J[r : \emptyset, H_d(r) \mid r \in J{\downarrow_v}, H(r) = \bar{\alpha}] = J$. Secondly, if $H(v) = \bar{\alpha}$ then nodes are reset. Since J is weakly winning, all nodes in $J{\downarrow_v}$ have hypothesis $\bar{\alpha}$. So all these nodes will be reset. □

Second, we show resetting the dependent nodes can be delayed as long as nodes are justified for the same player. Abusing notation, reachability for a set R in a justification J is denoted as $J{\downarrow_R} = \bigcup_{v \in R} J{\downarrow_v}$. Clearly $J{\downarrow_v} = J{\downarrow_{\{v\}}}$ which will allow merging adjacent reset operations according to the following lemma.

Lemma 3.9. *Given a possibly unsafe justification J and two sets R_1 and R_2 then*

$$\begin{aligned} &J[r : \emptyset, H_d(r) \mid r \in J{\downarrow_{R_1 \cup R_2}}, H'(r) = \bar{\alpha}] \\ &= J_{R_1}[r : \emptyset, H_d(r) \mid r \in J_{R_1}{\downarrow_{R_2}}, H(r) = \bar{\alpha}] \\ &\textit{where } J_{R_1} = [r : \emptyset, H_d(r) \mid r \in J{\downarrow_{R_1}}, H(r) = \bar{\alpha}] \end{aligned}$$

Furthermore, a reset operation for player $\bar{\alpha}$ commutes with justify operations for a node v and player α if the reset operation includes the justified node v.

Lemma 3.10. *Given a (likely not safe) justification J and a node v winning with dj for α and a set R where $v \in R$, then:*

$$\begin{gathered} J[r : \emptyset, H_d(r) \mid r \in J{\downarrow_R}, H(r) = \bar{\alpha}][v : dj, \alpha] \\ = \\ J_v[r : \emptyset, H_d(r) \mid r \in J_v{\downarrow_R}, H'(r) = \bar{\alpha}] \\ \textit{where } J_v \quad = J[v : dj, \alpha] \end{gathered}$$

Note how, after commuting the set of reset nodes is determined by the intermediate justification J_v. As such delaying the reset-phase can be implemented in-place, only using additional memory to represent the set R.

Proof. Let J_v be $J[v : dj, \alpha]$, let $J_{vR} = J_v[r : \emptyset, H_d(r) \mid r \in J_v{\downarrow_R}, H'(r) = \bar{\alpha}]$, let J_R be $J[r : \emptyset, H_d(r) \mid r \in J{\downarrow_R}, H(r) = \bar{\alpha}]$ and let J_{Rv} be $J_R[v : dj, \alpha]$.

We prove the hypothesis and outgoing edges of every node match in both justifications by classifying a node x in five cases:

- For $x = v$, v is assigned the direct justification dj and hypothesis α in J_{Rv} and J_v and, since $H_v(v) = \alpha$, also it was not reset in J_{vR}.
- When $x \neq v$, and $H_J(x) = \alpha$ then x is untouched, $J(x) = J_{Rv}(x) = J_{vR}(x)$ and $H_J(x) = H_{J_{Rv}}(x) = H_{J_{vR}}(x)$.
- If $x \notin J{\downarrow_R}$ then x is untouched as well and both $J(x) = J_{Rv}(x) = J_{vR}(x)$ and $H_J(x) = H_{J_{Rv}}(x) = H_{J_{vR}}(x)$.
- Nodes for which $x \in J{\downarrow_R}$, $x \neq v$, and $H_J(x) = \bar{\alpha}$ remain. Such a node x is reset in J_R and, since $v \in R$, in J_{vR}. As node x is not the assigned node v, it is unjustified in both J_{vR} and J_{Rv} with its default winner as hypothesis, $H_{J_{vR}}(x) = H_d(x) = H_{J_{Rv}}(x)$.

Thus J_{Rv} and J_{vR} are equal. □

Combining the three previous lemmas allows improving the implementation when justifying multiple nodes for the same player. After restating each **Justify** with Lemma 3.8 it is possible to use Lemma 3.10 commute the reset caused by v_1 and its assignment. Then Lemma 3.9 can be used to merge the reset part with that of v_2. Continuing to commute and merge alternatingly will delay these reset operations to the end, as shown in the following theorem:

Theorem 3.11. *When performing a sequence of* **Justify** *steps for nodes* $v_1, \dots v_n$ *for* α*, one can first perform all assignments and then perform all resets, i.e.,*

$$\begin{aligned}\textbf{Justify}(\textbf{Justify}(\dots(\textbf{Justify}(J, v_1, dj_1), v_2, dj_2), \dots), v_n, dj_n) \\ = J_{\bar{v}}[r : \emptyset, H_d(r) | r \in J_{\bar{v}}{\downarrow_R}, H(r) = \bar{\alpha}] \\ \textit{where } J_{\bar{v}} = J[v_1 : dj_1, \alpha] \dots [v_n : dj_n, \alpha] \\ \textit{and } R = \{v_1, \dots, v_n\}\end{aligned}$$

3.6 Safely modifying parity games

One of the uses for parity games is the verification of modal expressions. Strix [LMS20; MSL18] is the winner of the last reactive synthesis competition, SyntComp 2019. From a high-level view, Strix first transforms a LTL-expression to a parity game. An integrated solver then finds a solution for this parity game. The solution of this game is then transformed back to a solution for the LTL-expression. One of the improvements of Strix is generating the parity game at the same time as solving it. This allows the program to use multiple cores and allows stopping when a solution is found even if the parity game is only partially generated. However, a compatible solving algorithm is required: extending a parity game by adding edges and nodes must not break invariants of the solver. The algorithm used by Strix is strategy improvement, an algorithm that benchmarks as less-efficient.

This section shows how justification-based algorithms can adapt to a growing parity game. In particular, how to maintain a safe justification when parts of a parity game are modified. Since justification based algorithms are capable of continuing with any safe justification, this fulfils the requirements of Strix and allows for replacing the strategy improvement algorithm. A trivial solution exists for any algorithm: fully restarting the algorithm each time the game is extended. Thus the goal is to minimize the destruction a modification causes.

The following atomic changes are considered: adding and removing an edge, adding a node and removing a node without incoming edges, changing the owner

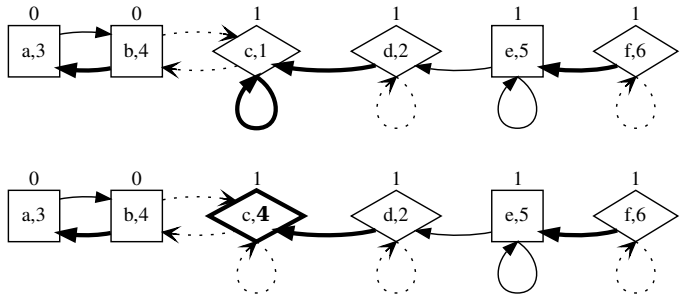


Figure 3.3: Above: The running example parity game with its solution.
Below: A different parity game with a single change: the priority of node c is now 4. Re-parametrising only the affected node results in an unsafe justification.

of a node, and changing the priority of a node. Each of these modifications may change the winner of the affected node, which may cascade to dependent nodes.

Definition 10 (A modification of a node in $\mathcal{PG}$)**.** *The parity game $\mathcal{PG}'$ is a modification of a node v in a parity game $\mathcal{PG}$ iff except for the node v the nodes, outgoing edges, owners and priorities of both game are equal.*

A naïve solution is simply removing the direct justification of the affected node(s), essentially adding the node as a parameter to fix the winner. The question that remains is how to proceed. As seen in Example 4, such a justification may be winning but unsafe. Thus violating the central invariant of justification-based algorithms and precondition of **Justify**.

Example 4. Figure 3.3 shows the result of modifying the priority of a node and re-parametrising only the affected node. The parity game and its solution (top figure) is modified by changing the priority of node c and removing the direct justification. The bottom figure shows the resulting unsafe justification. This justification is unsafe for two separate reasons. First, as the hypothesis of node c is not default, this violates the second condition of Definition 7. Second, as node f has priority 6 and as parameter node c with priority 4, the third condition is violated.

A solution can be found in **Justify**: if a node changes its hypothesis, all dependent nodes violate the weakly-winning invariant. In order to restore this invariant and keep the justification winning and safe, all dependent nodes are reset. The following proposition copies this approach by resetting all dependent nodes of a parity game modified at a node v:

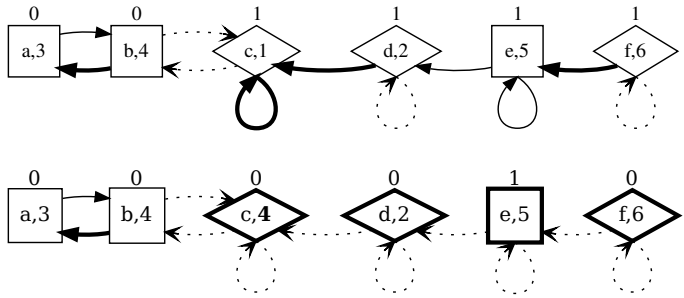


Figure 3.4: Above: The running example parity game with its solution. Below: The same modified parity game where the priority of node c is now 4. Re-parametrising all nodes reaching the affected node maintains a safe justification.

Proposition 3.12. *Let J be a safe justification for a parity game PG. Let $\mathcal{PG}'$ be a parity game that is a modification in $\mathcal{PG}$ of a node v. Then the justification J' is (allowing for adding or removing the node v) a safe justification for $\mathcal{PG}'$.*

$$J' = J[w : \emptyset, H_d(w) \mid w \in J{\downarrow_v}].$$

Example 5. Figure 3.4 shows how to maintain a safe justification by removing the direct justifications of all nodes dependent on the affected node. Note that in comparison to Example 4, node c has a default hypothesis and node f is a parameter itself.

3.7 Conclusion

This chapter defined a justification as a structure containing a sub-graph of the parity game graph and a hypothesis function determining the (assumed) winner of each node. Three accompanying properties (weakly-winning, winning and safe) further constrain justifications. Next, the action **Justify** is introduced to improve a safe justification by justifying one node. Proving the capability of **Justify** to eventually solve parity games results in a simple non-deterministic parity game solving algorithm.

Often this affects a number of other nodes which need to be reset. By using a graph-representation, resetting only the dependent nodes is easy.

We finished with two additional topics: Section 3.5 showed an important optimisation for justification. The ability to justify multiple nodes before resetting their dependent nodes allows for improving the cache behaviour of all implementations discussed in Chapter 5.

The second topic, discussed in Section 3.6, impacts the interaction with software that uses parity game solvers. With the exception of the strategy improvement solver in Strix, all solvers are one-shot: they solve one parity game. On the other hand, Strix generates and solves a parity game concurrently. Proposition 3.12 shows how to retain most of a justification while modifying the underlying game in various ways. This paves the way to solve a parity game with **Justify**-based algorithms instead of strategy improvement.

Chapter 4

Rebuilding recursive algorithms with justifications

The majority of this chapter is accepted for publication in [LBDar]. The second part of this paper presents the algorithms of Sections 4.1.1, 4.2, and 4.3. In Sections 4.1.2 and 4.4, justifications are applied to two extra algorithms.

This chapter shows how to rebuild basic versions of different well-known algorithms using **Justify**. Section 4.1 shows two variants of the nested fixpoint algorithm. Section 4.2, 4.3, and 4.4 show how to rebuild respectively Zielonka's algorithm, priority promotion, and tangle learning. Section 4.5 concludes this chapter.

In our versions, we represent the parity game G as (V, E, O, Pr) and the justification J as (V, D, H). Although all algorithms can start with any safe justification, the pseudocode only considers the safe empty justification $(V, \emptyset, H_d)$. The recursive algorithms operate on a subgame SG determined by a set of nodes V_{SG}. This subgame determines which **Justify**(J, v, dj) steps are considered to be performed on G. For convenience of presentation, G and its components V, E, O, Pr are considered global constants.

4.1 Nested fixpoint iteration

4.1.1 A naïve algorithm

Nested fixpoint iteration [BFL14; vR19; LBD20] is one of the earliest algorithms able to solve parity games. In Algorithm 4.1, we show a basic form that makes use of our **Justify** operation. It starts with the initial safe justification $(V, \emptyset, H_d)$. Iteratively, it determines the lowest priority p over all unjustified nodes, it selects a node v of this priority and justifies it. Recall from the proof of Lemma 3.5, that all unjustified nodes of this priority are justifiable. Every iteration increases the justification size. Eventually, all nodes are justified and a solution is obtained. For more background on nested fixpoint algorithms and the effect of justifications on the performance, we refer to Chapter 5 and our work in [LBD20].

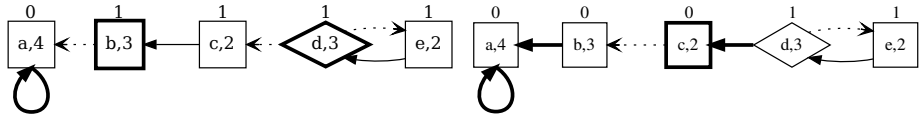
```
1  J_∅ ← (V, ∅, H_d)
2  J ← Fixpoint(G, J_∅)
3
4  Fn Fixpoint(G, J):
5      while J has unjustified nodes do
6          p ← min{Pr(v) | v is unjustified}
7          v ← an unjustified node with Pr(v) = p
8          dj ← a direct justification to justify v in J
9          J ← Justify(J, v, dj)
10     return J
```

Algorithm 4.1: A fixpoint algorithm for justifying nodes

4.1.2 Optimisations

Eliding the second precondition While implementing, it is important to note that the second precondition of **Justify** is trivial for the previous algorithm: Since p is the minimal unjustified priority all nodes have a greater or equal justification level. As a result, all direct justifications have $jl(dj) \geq p$ and the justification level condition can be removed. As calculating the justification level is linear w.r.t. the number of edges in the justification graph, obviating this precondition improves performance significantly.

Batching A second optimisation is batching: it is possible to justify multiple nodes of priority p every iteration instead of justifying one node. However,



(a) A parity game and justification with two justifiable nodes.

(b) The resulting non-winning justification (see nodes c and d) after disregarding the preconditions and justifying both.

Figure 4.1: Parity games for Example 6

justifying nodes should never interfere: Example 6 shows how justifying two justifiable nodes without re-verifying the preconditions may lead to an unsafe, even non-winning, justification.

Example 6 (Interfering justifications)**.** Figure 4.1a shows a parity game with a partial justification: J. In this justification node b is justifiable with $\{(b, a)\}$ for Even while node d is justifiable with $\{(d, c)\}$ for Odd. Justifying node b first invalidates the justification of node d with $\{(d, c)\}$ since node c is reset. Justifying node c first leaves node b justifiable. Subsequently justifying node b will reset nodes c, d and e. Both arrangements result in a safe justification graph.
With batching we would pre-compute the changes (set the direct justification of d to $\{(d, c)\}$, and set the direct justification of b to $\{(b, a)\}$ and reset c) and then execute them. However, as shown in Figure 4.1b, the result $J[b : \{(b, a)\}, 0][d : \{(d, e)\}, 1][c : \emptyset, 0]$ is a non-winning justification. In similar games, the justification can be winning but still violate condition (iii) of Definition 7.

Of course solutions exist: a first solution handles interference by resetting newly dependent nodes. Instead of recording the nodes to reset, only the nodes with their winning direct justification are recorded. After setting all direct justifications, the reset phase resets all dependent of all nodes that changed winner. Thus nodes d and e would be included in this reset phase. A flaw in this method is that this resets all newly dependent nodes even if an alternative direct justification exists. E.g., without the resetting node e, node d can still be justified with $\{(d, e)\}$.

Another solution avoids interference with a simple determination: Lemma 4.1 states justifying multiple nodes for the same player cannot interfere.

Lemma 4.1. *Given a safe justification J, a node v justifiable with dj_v for α and a different unjustified node w justifiable with dj_w also for α. Then w is justifiable in* $\mathbf{Justify}(J, v, dj_v)$ *with dj_w for α.*

Proof. Let J' be the justification $\mathbf{Justify}(J, v, dj_v)$, let H' be the hypothesis of J'. Since J is weakly winning, nodes are only reset if $H(v) = \bar{\alpha}$. Furthermore any reset nodes r have $H(r) = \bar{\alpha}$. Thus all nodes a with $H(a) = \alpha$ have $H'(a) = \alpha$ and $jl_{J'}(a) \geq jl_J(a)$. Thus the justification level of dj_w does not decrease. Finally, since node w is still unjustified in J' it remains justifiable. □

Algorithm 4.2 shows a variant which implements both optimisations: Line 8 collects all unjustified nodes with priority p that are justifiable for $\bar{\alpha}$ and justifies them in batch on Line 11. Line 8 also shows the simplified condition: an unjustified node v merely needs a winning direct justification instead of a justifying direct justification.

```
1  J_∅ ← (V, ∅, H_d)
2  J ← Fixpoint'(G, J_∅)
3
4  Fn Fixpoint'(G, J):
5      while J has unjustified nodes do
6          p ← min{Pr(v) | v is unjustified}
7          α ← p mod 2
8          E_ᾱ ← { (v, dj_v) | v is an unjustified node with Pr(v) = p and
                                dj is winning for node v and player ᾱ }
9          if E_ᾱ ≠ ∅ then
10             for (v, dj_v) ∈ E_ᾱ do
11                 J ← Justify(J, v, dj_v)
12         else
13             E_α ← { (v, dj_v) | v is an unjustified node with Pr(v) = p and
                                   dj is winning for node v and player α }
14             for (v, dj_v) ∈ E_α do
15                 J ← Justify(J, v, dj_v)
16     return J
```

Algorithm 4.2: A fixpoint algorithm for justifying nodes

A feature of nested fixpoint iteration is that it solves a parity game *bottom up*. It may take many iterations before it uncovers that the current hypothesis of some high priority unjustified node v is in fact wrong and so that playing to v is a bad strategy for α. The next algorithms are *top down*, they start out from nodes with the highest priority.

Merging multiple Justify steps A final optimisation uses Theorem 3.11 to exploit cache behaviour: Lines 10-11 (and analogously Lines 14-15) can be rewritten as

$$\begin{array}{ll} J & [v : dj_v, \alpha \mid (v, dj_v) \in E_{\bar{\alpha}}] \\ & [w : \emptyset, H_d(w) \mid w \in J{\downarrow_v}, (v, _) \in E_{\bar{\alpha}}, H(w) = \bar{\alpha}] \end{array}$$

By splitting the justification of nodes and the resets caused by these justifications the cache-pressure decreases since the nodes in $E_{\bar{\alpha}}$ are (after sorting by priority) adjacent.

4.2 Zielonka's algorithm

Zielonka's algorithm [Zie98], one of the oldest algorithms, is recursive and starts with a greedy computation of a set of nodes, called *attracted* nodes, in which the winner α of the top priority p has a strategy to force playing to nodes of top priority p.

In our reconstruction, Algorithm 4.3, attracting nodes is simulated at Line 7 by repeatedly justifying nodes v with a direct justification that wins v for α and has a justification level $\geq p$. Observe that the while test ensures that the preconditions of **Justify**(J, v, dj) on the justification level of v are satisfied. Also, every node can be justified at most once.

```
   input : A parity game G
1  J←Zielonka((V, ∅, H_d), V)
2
3  Fn Zielonka(J, V_SG):
4      p←max {Pr(v) | v ∈ V_SG}
5      α←p mod 2
6      while true do
7          while ∃v ∈ V_SG, dj : v is unjustified, v is justifiable with dj for α
               with jl(dj) ≥ p do
8              J←Justify(J, v, dj)
9          V_SSG←{v ∈ V_SG | jl(v) < p}
10         if V_SSG = ∅ then return J
11         J←Zielonka(J, V_SSG)
12         while ∃v ∈ V_SG, dj : v is unjustified, v is justifiable with dj for ᾱ
               with jl(dj) ≥ p + 1 do
13             J←Justify(J, v, dj)
```

Algorithm 4.3: A **Justify** variant of Zielonka's algorithm.

The procedure is called with a set V_{SG} of nodes of maximal level p that cannot be attracted by levels $> p$. It follows that the subgraph determined by V_{SG} contains for each of its nodes an outgoing edge (otherwise the opponent of the owner of the node would have attracted the node at a level $> p$) , hence this subgraph determines a parity game. The main loop invariants are that (1) the justification J is safe; (2) the justification level of all justified nodes is $\geq p$ and (3) $\bar{\alpha}$ has no direct justifications of justification level $> p$ to win an unjustified node in V_{SG}. The initial justification is safe and it remains so as every **Justify** call satisfies the preconditions.

After the attraction loop at Line 7, no more unjustified nodes of V_{SG} can be attracted to level p for player α. Then, the set of V_{SSG} of unjustified nodes of priority $< p$ is determined. If this set is empty, then by Lemma 3.5 all unjustified nodes of priority p are justifiable with a direct justification dj with $jl(dj) \geq p$, hence they would be attracted to some level $\geq p$ which is impossible. Thus, there are no unjustified nodes of priority p. In this case, the returned justification J justifies all elements of V_{SG}. Else, V_{SSG} is passed in a recursive call to justify all its nodes. Upon return, if $\bar{\alpha}$ was winning some nodes in V_{SSG}, their justification level will be $\geq p + 1$. Now it is possible that some unjustified nodes of priority p can be won by $\bar{\alpha}$ and this may be the start of a cascade of resets and attractions for $\bar{\alpha}$. The purpose of Line 12 is to attract nodes of V_{SG} for $\bar{\alpha}$. Note that **Justify**(J, v, dj) resets all nodes that depend on nodes that switch to $\bar{\alpha}$. When the justification returned by the recursive call shows that α wins all nodes of V_{SSG}, the yet unjustified nodes of V_{SG} are of priority p, are justifiable by Lemma 3.5 and can be won only by α. So, at the next iteration, the call to $Attr_\alpha$ will justify all of them for α and V_{SSG} will be empty. Eventually the initial call of Line 1 finishes with a safe justification in which all nodes are justified thus solving the game G.

Whereas fixpoint iteration first justifies low priority nodes resulting in low justification levels, Zielonka's algorithm first justifies nodes attracted to the highest priority. Compared to fixpoint iteration, this results in large improvements in justification size which might explain its better performance. However, Zielonka's algorithm still disregards certain opportunities for increasing justification size as it proceeds by priority level, only returning to level p when all sub-problems at level $< p$ are completely solved. Indeed, some nodes computed at a low level $i \ll p$ may have a very high justification level, even $+\infty$, and might be useful to revise false hypotheses at high levels, saving much work, but this is not exploited. The next algorithm, priority promotion, overcomes this limitation.

Merging Justify step As with nested fixpoint iteration, Theorem 3.11 is applicable to Lines 8 and 13. However there is a small problem with determining which nodes are justifiable: in both cases the while-condition requires a safe justification and a justification where resets are delayed is in almost all cases not safe. Thus, without modifications, resets can only be delayed in small batches.

In order to delay resets until all nodes are attracted, the first precondition of **Justify** is refined. One can approach this as follows: first define an α-delayed justification (J, α, R) where R is a set of nodes to be reset. The delayed justification can be consolidated as follows:

$$\mathit{Consolidate}((J, \alpha, R)) = J[w : \emptyset, H_d(w) \mid w \in J{\downarrow_v}, v \in R, H(w) = \bar{\alpha}]$$

Note that the $\mathit{Consolidate}((J, \alpha, \emptyset)) = J$. We define a delayed justification safe iff its consolidation is safe. Now it is possible to re-define **Justify** with for safe α-delayed justifications where dj wins v for α and keeping the second precondition:

$$\textbf{Justify}((J, \alpha, R), v, dj) = (J[v : dj, \alpha], \alpha, R \cup \{v\})$$

Using Lemma 3.8 and Theorem 3.11, correctness is proven with the following equality:

$$\textbf{Justify}(\mathit{Consolidate}(J, \alpha, R), v, dj) = \mathit{Consolidate}(J[v : dj, \alpha], \alpha, R \cup \{v\})$$

The resulting optimisation reduces the number of modifications ($J[\dots]$) of the justification graph: often, during attraction, a node is first reset and then attracted later. E.g., for nodes with a single outgoing edge. Moreover, generally, both attracting and resetting nodes account for over 90% of the run-time due to the arbitrariness of the game-graphs. So, reducing the number of modified nodes has a direct effect on efficiency.

4.3 Priority promotion

Priority promotion [BDM16b; BDM16a; BDM18] follows the strategy of Zielonka's algorithm except that, when it detects that all nodes for priority p are justified, it does not make a recursive call but returns the set of nodes attracted to priority p nodes as a set R_p to a previous level q. There R_p is added to the attraction set at level q and the attraction process is restarted. In the terminology of [BDM16b], the set R_p is a *closed p-region* that is *promoted* to level q. A *closed p-region* of V_{SG}, with maximal priority p, is a subset $R_p \subseteq V_{SG}$

that includes all nodes of V_{SG} with priority p and for which $\alpha = p \bmod 2$ has a strategy winning all infinite plays in R_p and for which $\bar{\alpha}$ cannot escape from R_p unless to nodes of higher q-regions won by α. We call the latter nodes the *escape nodes* from R_p. The level to which R_p is promoted is the lowest q-region that contains an escape node from R_p. It is easy to show that q is a lower bound of the justification level of R_p. In absence of escape nodes, R_p is promoted to $+\infty$.

```
   input : A parity game G
 1 J←(V, ∅, H_d)
 2 while ∃v ∈ V_G : v is unjustified do
 3 |  R_{+∞}← {v | jl(v) = +∞}
 4 |  V_SG←V \ R_{+∞}
 5 |  (J, _, _)←Promote(V_SG, J)
 6 |  while ∃v ∈ V_SG, dj : v is justifiable with dj and jl(dj) = +∞ do
 7 |  |  J←Justify(J, v, dj)
 8
 9 Fn Promote(V_SG, J):
10 |  p←max {Pr(v) | v ∈ V_SG}
11 |  α←p mod 2
12 |  while true do
13 |  |  while ∃v ∈ V_SG, dj : v is unjustified or jl(v) < p, v is justifiable
   |  |   with dj for α with jl(dj) ≥ p  do
14 |  |  |  J←Justify(J, v, dj)
15 |  |  R_p← {v ∈ V_SG | jl(v) ≥ p}
16 |  |  if Closed(R_p, V_SG) then
17 |  |  |  l←min{q | R_q contains an escape node of R_p}
18 |  |  |  return (J, l)
19 |  |  V_SSG←V_SG \ R_p
20 |  |  (J, l)←Promote(V_SSG, J)
21 |  |  if l > p then
22 |  |  |  return (J, l)
```

Algorithm 4.4: A variant of priority promotion using **Justify**.

Our variant of priority promotion is in Algorithm 4.4. Whereas **Zielonka** returned a complete solution J on V_{SG}, **Promote** returns only a partial J on V_{SG}; some nodes of V_{SG} may have an unfinished justification ($jl(v) < +\infty$). To deal with this, **Promote** is iterated in a while loop that continues as long as there are unjustified nodes. Upon return of **Promote**, all nodes attracted to the returned $+\infty$-region are justified. In the next iteration, all nodes with justification level $+\infty$ are removed from the game, permanently. Note that when promoting to some q-region, justified nodes of justification level $< q$ can

remain. A substantial gain can be obtained compared to the original priority promotion algorithm which does not maintain justifications and loses all work stored in J.

By invariant, the function **Promote** is called with a set of nodes V_{SG} that cannot be justified with a direct justification of level larger than the maximal priority p. The function starts its main loop by attracting nodes for level p. The attraction process is identical to Zielonka's algorithm except that leftover justified nodes v with $jl(v) < p$ may be rejustified. As before, the safety of J is preserved. Then the region R_p consists of elements of V_{SG} with justification level $\geq p$. It is tested (with **Closed**) whether R_p is a closed p-region. This is provably the case if all nodes of priority p are justified. If so, J, R_p and its minimal escape level are returned. If not, the game proceeds as in Zielonka's algorithm and the game is solved for the nodes not in R_p which have strictly lower justification level. Sooner or later, a closed region will be obtained. Indeed, at some point, a subgame is entered in which all nodes have the same priority p. All nodes are justifiable (Lemma 3.5) and the resulting region is closed. Upon return from the recursive call, it is checked whether the returned region ($R_{p'}$) promotes to the current level p. If not, the function exits as well (Line 22). Otherwise a new iteration starts with attracting nodes of justification level p for α.

We conclude this algorithm by noting that: i) Contrary to Zielonka's algorithm, there is no attraction step for $\bar{\alpha}$. Attracting for $\bar{\alpha}$ at p is the same as attracting for $\alpha' = \bar{\alpha}$ at $p' = p + 1$. ii) As with Zielonka's algorithm, the attraction-loop can be optimised to delay the reset operation.

4.4 Tangle learning

Tangle learning [van18a] extends priority promotion with two ideas: first, closed regions can be factored in multiple strongly-connected components (called tangles) and are promoted independently. Furthermore, instead of requiring that all nodes of a priority are justified, partial promotion is allowed: tangles can be found and promoted even if not all nodes of priority p are justified. Second, the tangles are saved. Under some conditions, stored tangles are re-usable, similarly to nodes tangles can be justified. This results in closing known cycles and increases the justification size in a way justifying a single node cannot. In this section, tangles are redefined for use with justifications:

Definition 11 (Tangle)**.** *An α-tangle $T \subseteq E$ is a set of edges containing the direct justifications for a set $U_T = \{v | (v, w) \in T\}$ such that: (i) The directed graph (U_T, T') with $T' = T \cap (U_T \times U_T)$ is non-empty and strongly connected;*

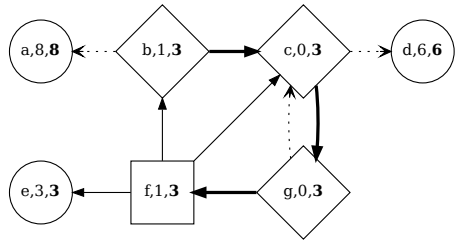


Figure 4.2: A part of a parity game containing a tangle. Square nodes are owned by Even, diamonds by Odd, circles fall outside the scope of this example. The labels inside a node are the name, priority and, in bold, the current justification level. The figure shows a tangle with 4 nodes (b, c, f and g) and 6 edges.

(ii) For each node $v \in U_T$ the outgoing edges $Out_T(v)$ form a direct justification to win v for α; (iii) All plays within (U_T, T') are won by α.

Abusing notation, the priority $Pr(T)$ of a tangle T is the maximal priority of nodes in U_T. The escapes Esc of a tangle T are the edges playing outside U_T, i.e., $Esc(T) = T \setminus T'$; the escapes form the direct justification of the tangle T. The justification level $jl(T)$ of a tangle T is $jl(Esc(T))$, i.e., $jl(Esc(T)) = min\{jl(w) | (_, w) \in Esc(T)\}$.

Observe that the priority of a tangle determines α: since tangles are strongly connected the winner of all cycles in the tangle has the same parity as the priority of the highest node. Furthermore, observe that only nodes owned by $\bar{\alpha}$ have escaping edges. A node owned by α, which has a single outgoing edge, must end in a node of U_T to be strongly connected. Finally, without the non-emptiness of T' every direct justification of a node for a player α would be an α-tangle.

Example 7. Figure 4.2 shows part of a parity game where the current justification has an Odd-tangle $T = \{(b, c), (c, g), (g, f), (f, b), (f, c), (f, e)\}$. The set of inner nodes is $\{b, c, f, g\}$, the only escape is node e. The priority of the tangle is 1.
The tangles appearance coincides with the jump in justification level from 1 to 3 when justifying node b or f, whichever was last.

Deriving tangles

Originally, tangles are extracted from a partial strategy during the process of solving a parity game. The procedure of finding a tangle consists of finding strongly connected components in a sub-graph of the game-graph. This sub-graph is determined by both the current region and the current partial strategy.

Since a justification replaces the partial strategy the same extraction method is usable: any safe justification can be partitioned in two graphs, one for each player, based on the hypothesis. In both graphs conditions (ii) and (iii) of Definition 11 are always fulfilled. Finding sub-graphs satisfying condition (i) remains. In essence this consists of finding strongly connected components. This is possible with several well-known algorithms such as Tarjan's algorithm [Tar72].

We define a procedure **Extract**(J, X) which returns the set of tangles in J with at least one node belonging to the set of nodes X. This procedure first runs the SCC-finding algorithm on J for all nodes in X. Then, the trivial SCC's (nodes without internal edges) are filtered. All remaining non-trivial SCC's are tangles to be returned by **Extract**.

Justifying tangles

Tangles are used to increase the justification size by overriding the direct justifications of (some of) the nodes in the tangle. As with regular nodes, the justification can only be modified if the necessary conditions are satisfied. In essence an α-tangle behaves as a node owned by $\bar{\alpha}$ with priority $Pr(T)$ and where Esc is the set of outgoing edges. The tangle is justifiable if all end-nodes in this set are won by α with a high enough justification level. The justification of a tangle justifies all nodes with lower justification levels in the tangle.

Definition 12. *An α-tangle T is justifiable in a safe justification J if (i) each escape-node e for which $(_, e) \in Esc(T)$ has $H(e) = \alpha$; (ii) the justification level $jl(T)$ of the tangle is larger than the priority of the tangle; and (iii) $jl(T)$ is larger than the justification level of some node in U_T.*

The justification of an α-tangle T in J is:

```
1 Fn TJustify(J, T):
2 |  S ← {v ∈ U_T | jl(v) < jl(T)}
3 |  for v ∈ S do
4 |  |  J ← Justify(J, v, Out_T(v))
5 |  return J
```

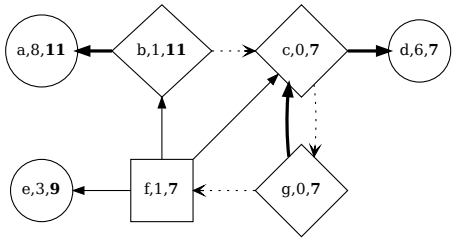


Figure 4.3: A different justification for the same part of the parity game as Figure 4.2 in which the previous tangle is justifiable.

However, Line 4 ignores the second precondition of **Justify**. Moreover, while often an ordering of nodes exists for which every operation is executable, as Example 8 will show, in rare cases none of the nodes in S can be safely justified.

The following example shows how justifying tangles works. Example 8 exhibits two special cases for working with tangles: (i) why nodes with a higher priority should not get reassigned (i.e., the necessity of $jl(v) < jl(T)$ for calculating the set O) and (ii) decomposing $\mathbf{TJustify}(J, T)$ in $\mathbf{Justify}(J, v, dj)$ steps can violate the second precondition of **Justify** and result in temporarily unsafe justifications. However, ignoring the second precondition while justifying all nodes v in U_T for which $jl(v) < jl(T)$ still results in a safe justification $\mathbf{TJustify}(J, T)$.

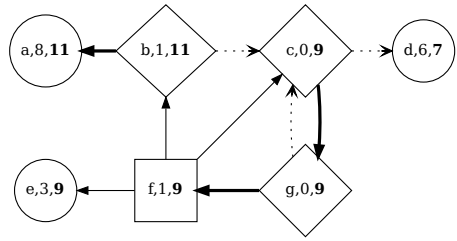


Figure 4.4: The result of justifying the tangle in Figure 4.3.

Example 8. Figure 4.3 shows a different situation compared to Example 7. In the current state the tangle of the previous example $T = \{(b,c),(c,g),(g,f),(f,b),(f,c),(f,e)\}$ is justifiable since (i) the only escaping node is won by Odd, (ii) $jl(T) = 9 > Pr(T) = 1$, and (iii) $jl(T) > jl(c) = jl(f) = jl(g) = 7$.
Justifying the tangle results in a new justification graph $\mathbf{Justify}(J,T)$ (See Figure 4.4). In this result the direct justification of node c is the singleton of (c,g), the direct justification of node g is the singleton of (g,f) and the direct justification of f is confirmed as $\{(f,b),(f,c),(f,e)\}$. The direct justification of node b is untouched. The new justification level of nodes c, f and g is 9. The justification remains safe iff J was safe.
This example shows two special cases that need to be accounted for. First, node b should not have its direct justification reassigned. Doing so would reduce the justification level of this node. This results in a smaller justification than J. Second, $\mathbf{TJustify}(J,T)$ cannot be decomposed in *increasing* $\mathbf{Justify}(J,v,dj)$ steps: Trying to justify node c with $T(c) = \{(c,g)\}$ results in an unsafe justification, while trying to justify node g with $T(g) = \{(g,f)\}$ does not increase the justification size. Both are disallowed by the second precondition of **Justify** used to guarantee correctness and termination. Ignoring this precondition results in the same justification graph:

$$\mathbf{TJustify}(J,T) = \mathbf{Justify}(\mathbf{Justify}(J,c,\{(c,g)\}),g,\{(g,f)\}$$

Since this circumvents the existing correctness proof, we will prove that **TJustify** restores safety at the end of the loop. First we use Lemma 3.10 to condense **TJustify** in a single operation:

$$\begin{aligned}
\mathbf{TJustify}(J,T) = \quad & J[v : T(v), \alpha \mid v \in S] \\
& [w : \emptyset, H_d(w) \mid w \in J{\downarrow_S}, H(w) = \bar{\alpha}] \\
\text{where } S = \quad & \{v \in U_T | jl(v) < jl(T)\} \\
J{\downarrow_S} = \quad & \textstyle\bigcup_{v \in S} J{\downarrow_v}
\end{aligned}$$

With this we prove, with four proofs, **TJustify** is safety-invariant and monotonic:

Proposition 4.2. *Let J be a safe justification, let T be an α-tangle justifiable in J. Then* **TJustify**(J,T) *is a safe justification larger than J.*

Before proving this proposition we recall the necessary definitions of weakly winning, winning and safe justifications: weakly winning states the outgoing edges of every justified node are a direct justification won by the hypothesis of that node. A justification is winning if it is weakly winning and if every infinite path in the justification is won by the hypothesis of the starting node. Finally, a justification is safe if it is winning, every parameter node has as hypothesis its default, and the justification level of every node is larger than or equal to its priority.

In the first proof proves justifying a justifiable tangle preserves weakly winning:

Proof. Let $J = (V, D, H)$, let T be an α-tangle justifiable in J, let $J' = (V, D', H') =$ **TJustify**(J,T), and let $S = \{v \in U_T | jl(v) < jl(T)\}$.

Consider a node v justified in J'. Remember, justified nodes have $J'(v) \neq \emptyset$. For v, we must prove $Out_{J'}(v)$ is a direct justification to win v under $H'(v)$. This consists of i) $J'(v)$ is a direct justification to win v for $H'(v)$ and ii) all w for which $(v,w) \in Out_{J'}(v)$ have $H'(w) = H'(v)$.

i) If v is in S then by Definition 12, $Out_{J'}(v) = Out_T(v)$. By condition (ii) of Definition 11, $Out_T(v)$ is a direct justification to win v for $\alpha = H'(v)$.

On the other hand, if v is not in S then $Out_{J'}(v) = Out_J(v)$ where $Out_J(v)$ is a direct justification that wins v for $H(v)$. As v is not reset, $H'(v) = H(v)$ and $J'(v)$ is a direct justification to win v for $H'(v)$.

ii) We prove all nodes v and $w \in Out_{J'}(v)$ (i.e., all edges $(v,w) \in J'$) have $H'(w) = H'(v)$. We consider five cases:

a) If $v \notin S$, and $H'(w) = H(w)$ then, since $(v,w) \in J$ and J is weakly winning: $H'(v) = H(v) = H(w)$.

b) If $v \notin S$, and $H'(w) \neq H(w)$ then v and w belong to $J{\downarrow_S}$ and $H(w) = \bar{\alpha}$ thus $H(v) = \bar{\alpha}$ and v is reset; a contradiction.

c) When $v \in S$ and $w \in S$ then $H'(w) = \alpha = H'(v)$ by the definition of **TJustify**.

d) When $v \in S$ and $(v, w) \in Esc(T)$, since T is justifiable, $H'(w) = H(w) = \alpha = H'(v)$.

e) The case when $v \in S$, $w \notin S$, and $(v, w) \notin Esc(T)$ remains: Since $v \in S$, the edge (v, w) originates from $T(v)$. Since $(v, w) \notin Esc(T)$, the edge (v, w) is an inner edge of T', thus w belongs to $U_T \setminus S$.

   Assume the internal node w has $H'(w) = H(w) = \bar{\alpha}$. We then build a unique parametrised play π in J starting in $\pi(1) = w$ and according to T: We note that two direct justifications for the same node winning for opposing players share exactly one edge. Since J is winning, for a justified node $\pi(i) \in U_T$ won by $H(\pi(i)) = \bar{\alpha}$, $Out_J(\pi(i))$ is a direct justification winning for $H(\pi(i)) = \bar{\alpha}$. Since T is an α-tangle, $Out_T(\pi(i))$ is a direct justification winning for α. Thus $Out_J(\pi(i)) \cap Out_T(\pi(i))$ is a singleton of the edge $(\pi(i), \pi(i+1))$. Since J is winning, $H(\pi(i+1)) = \bar{\alpha}$. Since T is a justifiable tangle, $\pi(i+1)$ is not an external (otherwise $H(\pi(i+1)) = \alpha$) nor unjustified (otherwise $jl(\pi(1)) \leq jl(\pi(i+1)) \leq Pr(T) < jl(T)$, thus $w \in S$). Thus $\pi(i+1)$ is a justified node in J and π is an infinite play within both J and T. Since J is winning, this play is won by $\bar{\alpha}$, since T is a tangle this play is won by α. This is a contradiction and thus $H'(w) = H(w) = \alpha = H'(v)$.

These five cases cover all possibilities. □

The second proof shows **TJustify** also preserves winning:

Proof. Let $J = (V, D, H)$, let T be an α-tangle justifiable in J, let $J' = (V, D', H') = \mathbf{TJustify}(J, T)$, and let $S = \{v \in U_T | jl(v) < jl(T)\}$.

Similarly to the proof for justifying nodes, only plays through nodes in S matter. Consider an infinite play with some tail π starting in a node v of S. Either π is fully in S and thus won by $\alpha = H(v)$. Or π escapes to a node w not in S, then w is in either $Esc(T)$ or in $U_T \setminus S$ and has $jl_J(w) \geq jl_J(T) > jl_J(v)$. By the definition of jl, the tail starting from w cannot return to S. In fact, it is a play in J which is won by $H(w) = \alpha$. Since J' is weakly winning, the initial play π is won by $H(v) = \alpha$ and, together with the previous proof, J' is winning. □

Thirdly, **TJustify** also preserves safety:

Proof. Let $J = (V, D, H)$, let T be an α-tangle justifiable in J, let $J' = (V, D', H') = \mathbf{TJustify}(J, T)$, and let $S = \{v \in U_T | jl(v) < jl(T)\}$. Every unjustified node in J' was either unjustified in J or had its justification removed

when justifying the tangle. Since J was safe and since reset nodes w have $H'(w)$ set to $H_d(w)$, all unjustified nodes v have $H'(v) = H_d(v)$.

It remains to prove all justified nodes v have $jl_{J'}(v) \geq Pr(v)$. Assume J' has a node v with $jl_{J'}(v) < Pr(v)$. Then this node must be in $J{\downarrow_S}$ otherwise $jl_{J'}(v) = jl_J(v)$. Since reset nodes have $jl_{J'}(v) = Pr(v)$, node v is also not reset. Thus v is a justified node with $H'(v) = \alpha$ with a path to an unjustified node w for which $Pr(v) > Pr(w)$.

Consider the path $v = v_1, \ldots, w = v_n$ from v to w in J'. Since J is safe, this is not a path in J. Since only nodes in S were justified, this path has one or more nodes of S. Let v_i be the last such node on the path. Node v_i is not the last node since all nodes in S are justified and v_n is not. Thus the tail $v_{i+1}, \ldots, v_n$ is a path in J and $jl_J(v_{i+1}) \leq Pr(w)$. Then node v_{i+1} is either in $U_T \setminus S$ or (v_i, v_{i+1}) is an edge in $Esc(T)$, in both cases $jl_J(v_{i+1}) \geq jl_J(T)$. This is a contradiction: $Pr(w) \geq jl_J(v_{i+1}) \geq jl_J(T) > jl_J(v) \geq Pr(v) > Pr(w)$. □

The fourth and final proof shows the resulting justification is larger than the initial justification J:

Proof. Let $J = (V, D, H)$, let T be an α-tangle justifiable in J, let $J' = (V, D', H') = \mathbf{TJustify}(J, T)$, and let $S = \{v \in U_T | jl(v) < jl(T)\}$.

To prove $J' >_s J$ we show i) all nodes v in S have $jl_{J'}(v) \geq jl_J(T) > jl_J(v)$, ii) the only nodes whose justification level decreased, the reset nodes r, have $jl_J(r) \leq jl(T)$, and iii) the justification level of all other nodes remains unchanged. Thus s_i increased for $max\{jl_{J'}(U_T)\}$ and for all $j > i$ the value s_j remains the same.

First, take a node v in S. By definition $jl_J(T) > jl_J(v)$. Consider any path π from v to some $w \in Par_{J'}(v)$ with $jl_{J'}(v) = Pr(w)$. Since all nodes in T are justified, $\pi(i) \in Esc(T)$ for some i and the tail $\pi(i), \ldots, w$ is a path in J. Thus $jl_J(\pi(i)) = jl_{J'}(\pi(i)) = Pr(w)$ and since $\pi(i) \in Esc(T)$, $jl(\pi(i)) \geq jl(T)$ and the justification level of v strictly increased, $jl_{J'}(v) > jl_J(v)$.

Second, all reset nodes r belong to $J{\downarrow_S}$ and r reaches some node v in J with $jl_J(v) < jl(T)$. Since J is safe, $jl_J(r) \leq jl_J(v) < jl_J(T)$.

Finally, no nodes in $J{\downarrow_S}$ and justified in J' decreased in justification level. Hypothetically, such node x would reach a reset node r and, since J is weakly winning, have $H_J(x) = H_J(r)$; this proves x is reset as well. The nodes v not in $J{\downarrow_S}$ remain: their parameters $Par(v)$ are unchanged, proving $jl_{J'}(v) = jl_J(v)$. □

input : A parity game G
$J \leftarrow (V, \emptyset, H_d)$
$\mathcal{T} \leftarrow \emptyset$
while $\exists v \in V_G : v$ *is unjustified* **do**
  $R_{+\infty} \leftarrow \{v | jl(v) = +\infty\}$
  $V_{SG} \leftarrow V \setminus R_{+\infty}$
  $(J, \mathcal{T}, _) \leftarrow$ **Tangle**$(V_{SG}, J, \mathcal{T})$
  while $\exists v \in V_{SG}, dj : v$ *is justifiable with* dj *and* $jl(dj) = +\infty$ **do**
    $J \leftarrow$ **Justify**(J, v, dj)

Fn Attract$_\alpha(J, p, I, \mathcal{T})$**:**
  while *true* **do**
    if $\exists v \in I$ *justifiable with* dj *for* α *with* $jl(dj) \geq p$ **then**
      $J \leftarrow$ **Justify**(J, v, dj)
      $I \leftarrow I \setminus \{v\}$
    else if $\exists T \in \mathcal{T}$ *justifiable with* $jl(T) \geq p$ *where* $U_T \cap I \neq \emptyset$ **then**
      $J \leftarrow$ **TJustify**(J, T)
      $I \leftarrow I \setminus U_T$
    else break
  return J
Fn Tangle$(V_{SG}, J, \mathcal{T})$**:**
  $p \leftarrow max\{Pr(v) | v \in V_{SG}\}$
  $\alpha \leftarrow p \bmod 2$
  while *true* **do**
    $I \leftarrow \{v \in V_{SG} | v$ is unjustified or $jl(v) < p\}$
    $J \leftarrow$ **Attract**$_\alpha(J, p, I, \mathcal{T})$
    $R_p \leftarrow \{v \in V_{SG} | jl(v) \geq p\}$
    $\mathcal{T}' \leftarrow$ **Extract**(J, R_p)
    if $\mathcal{T}' \neq \emptyset$ **then**
      $l \leftarrow max\{jl(T) | T \in \mathcal{T}'\}$
      return $(J, \mathcal{T} \cup \mathcal{T}', l)$
    $V_{SSG} \leftarrow V_{SG} \setminus R_p$
    $(J, \mathcal{T}', l) \leftarrow$ **Tangle**$(V_{SSG}, J, \mathcal{T})$
    if $l > p$ **then**
      return $(J, \mathcal{T}, l)$

Algorithm 4.5: A variant of on-the-fly tangle learning using **Justify**.

Bringing it together

So far we showed tangles can be found in and used to improve a justification. By combining tangles and justifications, it is possible to improve the previous

priority promotion algorithm. First, searching for tangles in the region replaces verifying if the region is closed. A node v in a p-region has $jl(v) > p$ iff it belongs to a tangle with priority p promotable to $jl(v)$. A closed region can be partitioned in tangles; every node belongs to a tangle with justification level larger or equal to the escape level of the region. Additionally, tangles often appear in unclosed regions: a single unclosed node in a region with a tangle suffices. Since this algorithm quickly reacts to cycles, an improvement can be expected compared to priority promotion. Second, attracting nodes is extended with justifying saved tangles. If some saved tangle is justifiable, then attracting it results in a strictly larger justification.

Applying these changes results in Algorithm 4.5. The differences to Algorithm 4.4 are small with three notable modifications: First, the procedure has an additional argument and variable $\mathcal{T}$ which stores all found tangles. Second, the node-attracting while-loop has been extracted to the procedure **Attract** (Line 25) and is extended to justifying tangles. Third, instead of testing if a region is closed, **Extract** (Line 27) finds all tangles in the region R_p and any found tangles are promoted.

4.5 Conclusion

This chapter showed how **Justify** can simulate multiple state-of-the-art algorithms. By generalizing the reconstructed algorithms it is possible to compare their strengths and weaknesses and heuristically explain their relative performance. The algorithms show a clear progression in complexity for deciding which node to justify next, leading to a larger expected increase in justification size: Nested fixpoint iteration has the simplest procedure, it justifies a single low-priority node. This algorithm is improved by justifying multiple nodes of the same priority for the same player before justifying the reset nodes. Zielonka's algorithm begins with attracting all nodes ordered by the increase in justification size they cause. Still, after evaluation nodes with large priorities will be ignored until all nodes with a lower priority are also justified. This is improved by priority promotion, which not only attracts nodes but also reacts when a region forms a quasi-dominion. The occurrence of a quasi-dominion indicates that re-evaluations of nodes of a larger priority may lead to improved justifications. Finally, tangle learning refines this promotion by factoring quasi-dominions into tangles, not only leading to better justifications but also saving them for re-use.

Note, however, that the steepest path, the shortest path and the fastest path of justifying nodes may be wildly different. While tangle learning may choose the largest increase in justification size, it may miss a simple solution. Even when

tangle learning uses the shortest path, extracting the tangles from a justification is a slower task than finding the unjustified node of lowest priority. Thus there is no absolute ranking of the different algorithms, only experimental results will show the costs and benefits.

4.5.1 Future work

Future research includes rebuilding other algorithms. One interesting algorithm is the quasi-polynomial algorithm of Parys [Par19]. Similarly to the previous algorithms it features attracting nodes. However the algorithm attracts to nodes which are neither justified nor default, i.e., its states include unsafe (but winning) justifications. Rebuilding this algorithm would thus involve relaxing the safety constraints on which the termination proof relies and reworking the original proof by Parys.

Another interesting path is analysing the similarity between the justification level and the measures used by progress measures algorithms [Fea+17; Jur00]. These algorithms assign a measure to every node, an array of nodes or priorities that count the length of the most advantageous path for one chosen player. Every step increases the measure of a single node until the path always includes a winning cycle (the length of the path exceeds the number of nodes). If no such paths remain, the procedure is repeated for the remaining player. For every node such paths exists for exactly one player thus determining the winners of each node and a corresponding strategy. We note the possibility of extending the justification level to a progress measure. In turn, resetting nodes is expected to become obsolete as progress measures increase monotonically. Furthermore, using the quasi-polynomial progress measure by Fearnley et al. [Fea+17] would lead to interesting quasi-polynomial implementations of state-of-the-art algorithms.

Chapter 5

Experimental evaluation of justification based algorithms

With the exception of Section 5.4, the content of this chapter is based on Lapauw et al. [LBD20] and was presented at VMCAI 2020. Section 5.4 continues the experiments with more algorithms.

5.1 Introduction

Chapter 4 showed how different algorithms can be rewritten to use justifications. Still, these algorithms are not equivalent to their original. A practical improvement is expected: the use of justifications should result in resetting fewer nodes.

This chapter will experimentally verify this hypothesis: Five new algorithms are implemented, tested and compared against existing state-of-the-art implementations. From this experiment, three conclusions are drawn: first, justifications generally improve the performance. Second, the representation of the justification graph influences performance. Third, adding justifications to more powerful base algorithms shows diminishing returns.

Section 5.2 details the setup of the experiments: the state-of-the-art solvers, benchmark and hardware used. This section compares three justification-based algorithms with the previous state-of-the-art. Section 5.3 shows the results of this experiment. Section 5.4 then repeats the experiment with more

algorithms: four justifications algorithms are compared to the updated state-of-the-art. Section 5.5 analyses the data from another perspective. Separating the experiments in crafted and random parity games yields some additional insights. Finally, Section 5.6 summarises the effects of adding justifications as seen in the experiments.

5.2 Experimental setup

In the literature, we selected three parity game solving projects:

- PGSolver [FL09a; FL09b] (`github.com/tcsprojects/pgsolver`) is a collection of tools for solving parity games written in OCaml. We refer to its algorithms with `pgsolver-*`.
- Oink [van18b; van18a] (`github.com/trolando/oink`) is a recent parity game solver suite written in C++. We refer to its algorithms with `oink-*`.
- pbespgsolver (`www.mcrl2.org`) is part of the mCRL2 toolset [Cra+13], centred around formal verification of automata. We refer to its algorithms with `pbes-*`.

Each project implements several algorithms, among them are fixpoint iteration (referred as `*-fp`, Zielonka's algorithm (referred as `*-zlk`), priority promotion (referred as `*-pp`) and tangle learning (referred as `*-tl`). For example, with `oink-tl` we refer to the tangle learning algorithm in the Oink solver. In total, 13 algorithms are benchmarked, 8 from the above three solvers and 5 algorithms we implemented. The implemented algorithms are `prty-fp`, `prty-fp-just`, `prty-tl-just`, `prty-tl`, `oink-zlk-just` and `oink-pp-just`. All `*-just` implementations use Lemma 3.10 as optimisation. All `prty-*` solvers are implemented in Rust and are available at `bitbucket.org/krr/prty`.

Two implementations, `oink-zlk-just` and `oink-pp-just`, are modifications of implementations in Oink where justifications were added with minimal changes to the original code. `oink-zlk-just` implements Algorithm 4.3 and `oink-pp-just` implements Algorithm 4.4.

The fixpoint iteration algorithm is only implemented in PGSolver, which, as Table 5.1 shows, has the worst performance of all three solvers. So it seems as a bad idea to modify that implementation. Therefore, we implemented our own version of the fixpoint algorithm (`prty-fp` implements Algorithm 2.2) and an extension with justifications (`prty-fp-just` implements Algorithm 4.2).

As for tangle learning, extending the Oink implementation was too involved: tangle learning is significantly more complex than Zielonka's algorithm. We started from scratch, implementing a baseline (`prty-tl`, Algorithm 2.7) and an extension with justifications (`prty-tl-just`, Algorithm 4.5). `prty-tl` and `oink-tl` perform similarly across all statistics: timing, number of iterations, attracted nodes.... This is explained by a few factors: first, both C++ and Rust are compiled languages. Second, the major hotspots are I/O bound (parsing[1]) or memory bound (attracting nodes, Tarjan's algorithm). There is no CPU-bound due to the lack of intensive arithmetic and a lack of predictable memory accesses.

However two noticeable differences were found: First, `prty-tl` and `prty-tl-just` use less memory during the preparation of the parity game. This leads to an increase in games solved within the memory limit minimally affecting solving time. This improvement has since been adapted for Oink as well (See Section 5.4). Second, `prty-tl` and `prty-tl-just` use a watch-structure for attracting nodes instead of counting. As both finding a new watch or re-counting occur in cold programming code the difference is negligible[2].

In literature, we found information on two benchmarks. The parity game benchmark [Kei15] and one with large random graphs used for benchmarking tangle learning [van18a]. These benchmarks[3] have been executed for the selected algorithms with a time-limit of 100 seconds and a memory limit of 6 GB on a PC with an Intel 'i7-4770' CPU.

5.3 Results

A summary of the results for these benchmarks is reported in Table 5.1: the number of games solved within 1 second, 10 seconds, and within the time-limit. The par2 score is also reported, it is the sum of the run-times of all solved games and a penalty of twice the time-limit (200 seconds) for each unsolved game. The results for the existing algorithms are comparable to those of [van18a] and [van18b].

This benchmark ranks the base algorithms as follows: tangle learning performs best, priority promotion performs better than Zielonka's algorithm and fixpoint iteration is the worst algorithm. The frameworks can also be ranked by

[1]Less than ten games are large enough for parsing to take more than 1 s. The parsing speed of all algorithms is similar and due to the density these games are trivial to solve. Thus the effects of parsing on the benchmark was considered insignificant.

[2]Combining watches and tangles complicates the code sufficiently so as to be avoided.

[3]The benchmarks can be reproduced in the VMCAI 2020 virtual machine (10.5281/zenodo.3533104) with the artifact at 10.5281/zenodo.3510292.

comparing different implementations of the same algorithm: Oink performs best, then pbespgsolver second-best and PGSolver ranks last. More interesting is the effect of adding justifications to various algorithms. The table shows that each justification variant performs better than the corresponding base algorithm.

The fixpoint algorithm `prty-fp-just` outperforms `prty-fp` with 289 additional solved instances. As it ranks far below the best algorithms, we do not analyse it in more detail but focus on the other implementations with justifications.

Figure 5.1 is a cactusplot of the 100 most time consuming instances; it zooms in on the top six algorithms. The figure shows that adding justifications results in a clear performance inprovement for Zielonka's algorithm (`oink-zlk` vs `oink-zlk-just`) and tangle learning (`prty-tl` vs `prty-tl-just`). The figure shows a general improvement for both `oink-zlk-just` and `prty-tl-just`. Both the table and the figure show that our baseline tangle learning algorithm, `prty-tl`, performs better than `oink-tl`. However both the table and the figure show a difference in instances solved between `prty-tl` and `oink-tl`. Further analysis learned us that the cause is a solvable memory inefficiency in Oink. To control this variable, we consider the 1277 instance solved by both versions: `oink-tl` needs 2626 seconds to solve them while our baseline, `prty-tl`, needs 2541 seconds. See Section 5.4 which repeats this experiment with an updated version of Oink.

Configuration	Both benchmarks (1297 instances)			
100s / 6GB	<1s	<10s	<100s	par2 (s)
prty-tl-just	976	1215	1289	4912
prty-tl	959	1207	1288	5729
oink-tl	947	1209	1277	7193
oink-zlk-just	947	1206	1260	10464
oink-pp	952	1188	1245	13295
oink-zlk	932	1186	1239	14471
pbes-pp	919	1141	1205	21406
pbes-zlk	860	1072	1140	34437
prty-fp-just	863	1094	1134	34866
pgsolver-pp	698	1034	1138	36453
pgsolver-zlk	625	910	1035	56824
prty-fp	590	780	845	92529
pgsolver-fp	532	743	807	100616

Table 5.1: Number of solved instances in less than 1, 10, and 100 seconds and par2 score (lower is better), sorted by the par2 score

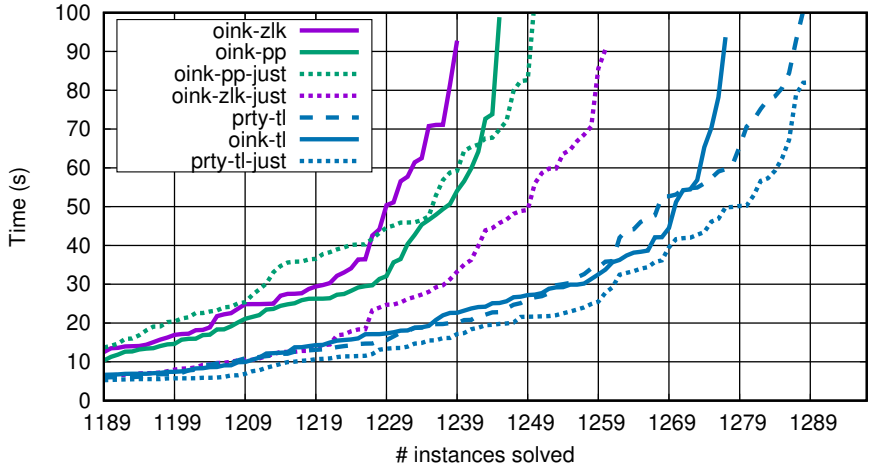
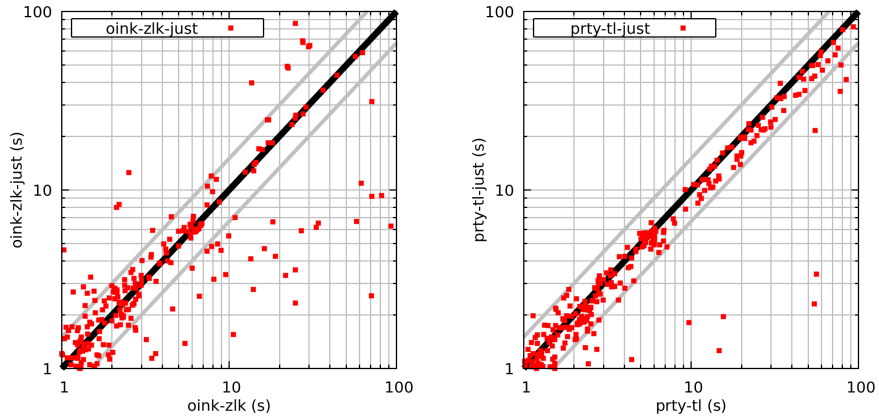


Figure 5.1: Cactusplot showing runtime of the slowest 100 parity games for the top six algorithms



(a) Comparison for Zielonka's algorithm

(b) Comparison for tangle learning

Figure 5.2: Comparisons of run times

Figure 5.2 zooms in on the differences in run-time for all instances solved by both versions of an algorithm. Below the black line are instances for which the justification version is faster; above the line those for which the justification version is slower. Figure 5.2a compares `oink-zlk` with `oink-zlk-just`. While

instance	runtime (s)		# attracted nodes	
	prty-tl	-just	prty-tl	-just
jurdzinskigame(500,50)	2.8	0.7	663k	51k
jurdzinskigame(500,100)	7.2	1.3	2.6M	101k
jurdzinskigame(100,500)	9.4	1.9	13M	101k
jurdzinskigame(500,200)	28.6	2.0	10M	201k
jurdzinskigame(200,500)	30.2	3.3	25M	201k
jurdzinskigame(500,500)	(190)	7.7	63M	502k
ABP_Onebit_(3,1,1,weak-bisim)	70.6	66.7	177M	40M
rn-1000000-1000000-1-2-4	75.8	58.4	163M	157M
rn-1000000-1000000-1-2-9	79.0	62.6	168M	164M
SWP(4,3,infinitely_often_rw)	79.1	78.9	104M	27M
rn-700000-700000-1-2-5	94.6	81.9	217M	235M
rn-700000-700000-1-2-4	99.4	81.4	224M	220M

Table 5.2: The reduction of runtime is coupled to a reduction in number of attracted nodes

there are examples with a large speed-up, there are also examples with a significant slowdown. We assume that most of the variability is caused by the explicit representation of the justification graph that uses hashing for direct access to the inverse of the justification function. Indeed, performance analysis showed that a major chunk of time is spent maintaining this data structure. Overall, there are more examples with a significant speed-up than with a significant slowdown and the overall time for solving the 1233 common examples is reduced from 2629 s to 2293 s. Considering that `oink-zlk-just` solves 21 extra instances, we conclude that adding justifications improves Zielonka's algorithm.

As the explicit justification graph used in Zielonka's algorithm caused a lot of variability in the run-time of individual instances, we opted for an implicit representation of the justification graph in tangle learning; it derives the justification graph from the existing information about nodes. Figure 5.2b, which compares `prty-tl` with `prty-tl-just`, indeed shows much less variability and we see, for instances requiring more than 10 s, a modest but almost consistent improvement. The overall time to solve the 1277 instances is reduced from 2541 s to 2159 s which is a 15% improvement.

The figure also shows a few examples with a significant speed-up while none with a significant slowdown. It turns out all these instances are so called Jurdziński games [Jur00]. A detailed profiling learned us that up to 80% of the time is spent on attracting nodes and that justifications reduce the number of attracted and reset nodes in Jurdziński games with several orders of magnitude.

Table 5.2 shows details about the Jurdziński game instances as well as some other hard instances. It also includes the 500 by 500 Jurdziński game, the most difficult Jurdziński game in the data set, which is not part of Figure 5.2b as it requires more than 100 s to solve. In this game, the number of attracted nodes is reduced from 126M to 501k nodes and the run-time from 190 seconds to less than 8 seconds. The other instances in Table 5.2 are representative for the instances in the top right of Figure 5.2b. Justifications reduce the number of attracted nodes by only a fraction of the total. Still, there are a small improvements in run-time. We conclude that justifications are a significant improvement for tangle learning.

5.4 Extended experimental results

In light of the previous results, additional experiments were performed: first, Oink received a major update, which reduced its memory usage. This was one of the two noticeable differences between prty and Oink. As before, we will write `oink`^{m}`-*` for algorithms based on this update. Second, implementing justified tangle learning showed a simpler graph structure performs better. This idea was used to further improve `oink`^{m}`-zlk-just`. Third, `oink`^{m}`-pp-just` implements priority promotion with justifications. Finally, the update of Oink includes three new fixpoint induction algorithms that were benchmarks as well. In total, ten new implementations in Oink are discussed. The results together with a selection of previous benchmarks can be found in Table 5.3.

`oink`^{m}`-tl` reduces the memory usage during preprocessing compared to `oink-tl`. Two additional benchmarks were run for tangle learning: `oink`^{m}`-tl` as the suggested state-of-the-art configuration and, since `prty-tl` and `prty-tl-just` lack preprocessing optimisations, the same algorithm with these optimisations disabled, `oink`^{m}`-tl-noopt`.

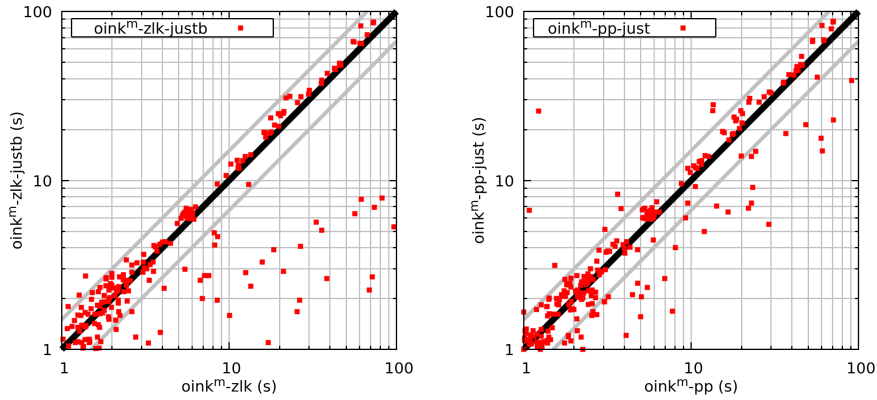
The table shows that the gap between `oink`^{m}`-tl` and `prty-tl` has been closed. In fact, `oink`^{m}`-tl` is now faster and more memory efficient than both `prty-tl` and `prty-tl-just`: it solves two more instances for which `prty-tl` exceeds the memory-limit. Comparing with `oink`^{m}`-tl-noopt` indicates i) optimisations still have some effect and should further improve `prty-tl-just`, and ii) the choice of programming language (C++ compared to Rust) has little influence. Comparing Table 5.1 and Table 5.3 show the same improvements also apply to `oink`^{m}`-zlk`, `oink`^{m}`-zlk-just` and `oink`^{m}`-pp` compared to `oink-zlk`, `oink-zlk-just` and `oink-pp`.

Configuration:	Both benchmarks (1297 instances)			
100s / 6GB	<1s	<10s	<100s	par2 (s)
oink^m-tl	976	1211	1290	4862
prty-tl-just	976	1215	1289	4912
oink^m-tl-noopt	967	1208	1289	5346
prty-tl	959	1207	1288	5729
oink^m-zlk-justb	981	1220	1282	6411
oink^m-pp-just	962	1198	1279	7645
oink^m-zlk-just	945	1210	1267	9377
oink^m-pp	974	1193	1262	10477
oink^m-zlk	978	1190	1255	11749
oink^m-fpjg	889	1079	1145	33591
prty-fp-just	863	1094	1134	34866
oink^m-fpj	867	1063	1126	37364
oink^m-fpi	867	1032	1080	45976
prty-fp	590	780	845	92529

Table 5.3: Number of instances solved by the additional algorithms in less than 1, 10, and 100 seconds and par2 score (lower is better), sorted by par2 score.

`oink`^{m}`-zlk-justb` is a second variant of `oink`^{m}`-zlk-just`. Optimising `prty-tl-just` has shown that a simpler data-structure performs better: `oink`^{m}`-zlk-just` maintains, for each node, a set of incoming edges. In theory, this representation has the best time-complexity for iterating all dependent nodes. However, in practice, most difficult parity games are sparse and there are on average only a few dependent nodes. As such, this representation incurs significant overhead for both time and memory. In contrast, `oink`^{m}`-zlk-justb` pre-calculates a static overestimation, namely, a list of all possible incoming edges for each node. Finding the dependent nodes then iterates over this list and filters all nodes which are currently not dependent. The results show a further improvement of solving 15 instances. Figure 5.3a shows a similar comparison as Figure 5.2a. For `oink`^{m}`-zlk-justb` the improvement is more consistent: there are far less extreme regressions.

`oink`^{m}`-pp-just` is a variant of `oink`^{m}`-pp` with justifications similar to Algorithm 4.4. Again using justifications improves the number of games that can be solved. From the onset, the algorithm used the same static overestimation as `oink-tl`, `oink`^{m}`-zlk-justb` and others. For completeness, Figure 5.3b shows the same comparison as the other pairs. A similar consistent improvement can be seen.



(a) Comparison for Zielonka's algorithm (b) Comparison for priority promotion

Figure 5.3: Comparisons of run times (extended results)

`oink`^{m}`-fpi`, `oink`^{m}`-fpj` and `oink`^{m}`-fpjg` are three new fixpoint induction algorithms included in the new version of Oink. `oink`^{m}`-fpi` is an implementation of the fixpoint algorithm with freezing by van Dijk and Rubbens [vR19]. This implementation improves the solving time of the basic fixpoint algorithm. As opposed to the approach by Friedmann and Lange [FL09a], it is capable of calculating a winning strategy without using memory proportional to the solving-time. `oink`^{m}`-fpj` and `oink`^{m}`-fpjg` are two independent implementations of the justified fixpoint algorithm by van Dijk. As the results show: using justifications helps to solve more instances. `oink`^{m}`-fpjg` performs best due to the preprocessing options Oink offers.

5.5 Results split w.r.t. benchmark

The type of parity games provided by the two benchmarks have large differences. From the benchmark collected by Keiren [Kei15], the crafted instances are used. In contrast, large random games of [van18a] are sparse and lack predetermined structure. This section analyses how much the differences in the games influence the standings of the algorithms.

Tables 5.4 and 5.5 show the individual results per benchmark. Both tables show the par2-score, the total runtime of the successful runs in seconds, and the number of failures of the relevant solvers. The tables are sorted by par2-score.

Solver	par2 (s)	runtime(s)	# fail (of 1037)
oink^m-pp	2240	2240	0
oink^m-ppj	2883	2483	2
oink^m-tl	3029	2029	5
prty-tl-just	3138	1938	6
prty-tl	3672	2272	7
oink^m-zlkjb	4490	2490	10
oink^m-fpjg	4569	2369	11
oink-pp	4846	1646	16
oink-tl	5287	1687	18
oink^m-zlk	5510	2310	16
oink^m-zlkj	7722	2322	27
oink^m-fpj	7879	2279	28
oink-zlk	8192	1792	32
prty-fp-just	11648	1048	53
prty-fp	40530	2130	192

Table 5.4: PG-bench crafted runs

Table 5.5, the crafted games, shows `oink`^{m}`-pp` performs best, having no failures. This is significantly different from the previous version, `oink-pp`, which had 16 failures. All updated `oink`^{m}`-*` solvers display similar improvements.

The results for adding justifications to different algorithms are mixed for this benchmark: while most justification-based variants solve the benchmark better, `oink`^{m}`-pp-just` and `oink`^{m}`-zlk-just` perform worse.

Table 5.5, the large random graphs, shows all justification based algorithms outperforming the original. Surprisingly, `oink`^{m}`-zlk-just` takes first place in this benchmark, despite the penalty for solving less runs. The results of `oink`^{m}`-zlk-justb` show a similar speed, outperforming `prty-tl` despite solving three runs less.

As expected, the update of Oink did not greatly affect the results of the algorithms for this benchmark. The update focused on improving memory-consumption and all large random games easily fit in the provided memory. However, we still note a small reduction in runtime.

We conclude that for crafted parity games, priority promotion performs best, while for random games, Zielonka's algorithm is surprisingly efficient. Still, in line with the previous sections, we expect tangle learning will perform better than Zielonka's algorithm for more difficult random games.

Solver	par2 (s)	runtime(s)	# fail (of 260)
oink^m-zlkj	1655	1055	3
prty-tl-just	1774	1374	2
oink^m-tl	1832	1432	2
oink-tl	1905	1505	2
oink^m-zlkjb	1922	922	5
prty-tl	2056	1656	2
oink^m-ppj	4762	1562	16
oink^m-zlk	6239	1039	26
oink-zlk	6280	1080	26
oink^m-pp	8237	1237	35
oink-pp	8450	1249	36
prty-fp-just	23218	1218	110
oink^m-fpjg	29023	823	141
oink^m-fpj	29485	885	143
prty-fp	52000	0	260

Table 5.5: Large random graphs

5.6 Conclusion

Our experiments demonstrate that adding a justification graph to a parity game solver algorithm is beneficial to the performance. Implementing multiple algorithms, from fixpoint induction to tangle learning, shows the improvement by using justifications generalises. As such we expect more experiments that test other justification-based algorithms.

Furthermore, the additional experiments confirmed the some remaining hypotheses of Lapauw et al. [LBD20]: i) adding justifications to priority promotion also improves this algorithm, ii) a well-chosen data-structure further improves Zielonka's algorithm with justifications.

Finally, we have shown that `prty-tl-just`, a tangle learning solver with justifications, improved the state-of-the-art for this benchmark. However, there are clear diminishing returns: the more complex the algorithm the smaller the improvement. We presume more complex algorithms also use more information derived from the partial strategy. It is used in Zielonka's algorithm to remove attracted nodes, in priority promotion to determine whether a region is closed, and in tangle learning to find tangles.

Still, it is risky to extrapolate diminishing returns as the benchmark is almost exhausted: the best algorithm solves 1290 out of 1297 instances and solves all

instances in 400 seconds with 15 GB of memory. Furthermore, only 82 instances need more than 10 seconds to solve. On the other hand, van Dijk [van19] shows there exist small parity games which execute an exponential number of iterations. For these games, Zielonka's algorithm, priority promotion, or tangle learning, all have an exponential runtime. Thus, better control over the difficulty of a parity game without significantly changing the game size is of interest.

Chapter 6

An equivalence between propositional nested fixpoint definitions and parametrised parity games

The inspiration for applying justification [DD93] to parity games was found in Hou and Denecker [HD09]. This paper describes a logic FO(FD) which is first-order logic extended with nested fixpoint definitions and its propositional calculus subset PC(FD). By describing the models of an PC(FD) expression a computational problem emerges: SAT(FD) is the satisfiability of propositional fixpoint definitions PC(FD). Two approaches for solving this problem have been implemented: Hou and Denecker [HD09] use justifications in a domain-specific SAT modulo theory. Another approach [HDD10] encodes the problem in difference logic, which is then solved by an SMT-solver implementing difference logic. In turn, justifications were previously used by Mariën et al. [Mar+08] for solving a simpler logic called satisfiability with inductive definitions, SAT(ID).

This chapter shows how justifications for PC(FD) are equivalent to those defined in Chapter 3 for parity games. After introducing the preliminaries (lattices and propositional calculus), the PC(FD) logic syntax and semantics are defined. Expressions in PC(FD) are called Nested Fixpoint Definition (NFD). Section 6.3 specifies a surjection for transforming a nested fixpoint definition to a parity game. The section ends with proving semantics are conserved: a solution of transformed parity game can be transformed to a model of the original

NFD. Finally, Section 6.4 details a right inverse of the previous transformation. Transforming a NFD to a parity game and back results in a equivalent canonical NFD.

This chapter shows not only linear transformations exist between both decision problems, but the equivalency also extends to the witnesses: winning strategies and witness justifications are polynomial witnesses for either decision problem. We conclude in Section 6.5 that more techniques, algorithms and applications can be transferred and we present some possibilities.

6.1 A positive fragment of Propositional Logic

A propositional logic starts with an expression over a set of symbols: a finite set Σ of propositional atoms, braces "(" and ")", and logical connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or). A positive formula is an expression with these symbols with a specific form:

Definition 13 (Positive formula [HDD10])**.** *A positive formula ϕ over a propositional vocabulary Σ is defined inductively as:*

- *any propositional atom $q \in \Sigma$ is a positive formula*
- *if ϕ and ψ are positive formulae, then $(\phi \wedge \psi)$ is a positive formula*
- *if ϕ and ψ are positive formulae, then $(\phi \vee \psi)$ is a positive formula.*

Definition 14 (Propositional formula)**.** *A propositional formula ϕ over a propositional vocabulary Σ is defined inductively as:*

- *any propositional atom $q \in \Sigma$ is a propositional formula,*
- *if ϕ is a propositional formula, $\neg\phi$ is a propositional formula,*
- *if ϕ and ψ are propositional formulae, $(\phi \wedge \psi)$ is a propositional formula,*
- *if ϕ and ψ are propositional formulae, $(\phi \vee \psi)$ is a propositional formula.*

Thus the set of positive formulae is a subset of all propositional formulae. For convenience, we give conjunction $\wedge$ precedence over disjunction $\vee$. As both are associative, often braces can be removed. For example: $(((q \vee r) \wedge s) \vee t)$ is simplified to $(q \vee r) \wedge s \vee t$.

Before introducing the semantics of a positive formula, we define the truth values. We denote the truth values *true* as $\mathbf{t}$ and *false* as $\mathbf{f}$. We define truth

order $\leq$ for the set $\{\mathbf{f}, \mathbf{t}\}$ as the total order induced by $\mathbf{f} < \mathbf{t}$. We define the inverse $\mathbf{t}^{-1}$ as $\mathbf{f}$ and $\mathbf{f}^{-1}$ as $\mathbf{t}$.

In this chapter, we use the two-valued semantics for propositional formulae. First, we define an interpretation for the vocabulary Σ. Next, given such interpretation, we define the evaluation of a propositional formula over a matching vocabulary.

Definition 15 (Interpretation [HDD10])**.** *Given a vocabulary Σ, an interpretation I over Σ is a function mapping propositional atoms of Σ to a truth value in $\{\mathbf{t}, \mathbf{f}\}$. We define a partial order comparing interpretations I and I' over the same vocabulary Σ: $I \leq I'$ iff for every atom q in the vocabulary $I(q) \leq I'(q)$ holds. This partial order forms a complete lattice for interpretations over a fixed vocabulary Σ. The minimal interpretation, denoted with $\bot$, maps all atoms to $\mathbf{f}$, the maximal interpretation, denoted with $\top$, maps all atoms to $\mathbf{t}$.*

Hou et al. [HDD10] also define addition of two interpretations. Given two interpretation I and I' over two disjoint vocabularies resp. Σ and Σ', we denote with $I + I'$ the interpretation over $\Sigma \cup \Sigma'$ which maps all atoms q of Σ to $I(q)$ and all atoms q' of Σ' to $I'(q')$. Given a subset vocabulary Σ' of Σ, the restriction of an interpretation I over Σ, $I|_{\Sigma'}$, denotes the interpretation mapping atoms q of Σ' to $I(q)$.

Definition 16 (Evaluation of propositional formula)**.** *Given a propositional formula ϕ and an interpretation I, both over a vocabulary Σ. The evaluation ϕ^I is a truth value calculated as follows:*

- *if ϕ is a propositional atom q in Σ then $\phi^I = I(q)$,*
- *if ϕ is a negation $\neg\psi$ then $\phi^I = (\psi^I)^{-1}$,*
  i.e., ϕ is $\mathbf{t}$ iff ψ is $\mathbf{f}$,
- *if ϕ is a conjunction $(\psi \wedge \chi)$ then $\phi^I = min_{\leq}\left\{\psi^I, \chi^I\right\}$,*
  i.e., ϕ is $\mathbf{t}$ iff both ψ and χ are $\mathbf{t}$,
- *if ϕ is a disjunction $(\psi \vee \chi)$ then $\phi^I = max_{\leq}\left\{\psi^I, \chi^I\right\}$,*
  i.e., ϕ is $\mathbf{f}$ iff both ψ and χ are $\mathbf{f}$,

As every positive formula is propositional, positive formulae can be evaluated as well. Furthermore, since both $\wedge$ and $\vee$ are monotonic, all positive formula are monotonic: if $I \leq I'$ then $\phi^I \leq \phi^{I'}$.

6.2 Nested fixpoint definitions

Most of the definitions in this section are as defined by Hou [HD09; HDD10; Hou10].

6.2.1 Syntax

Definition 17 (Rule [HDD10])**.** *A rule over Σ is an expression of the form:*

$$(q \leftarrow \phi)$$

where q is a propositional atom in Σ and ϕ is a positive formula over Σ. The atom p is known as the (defined) head of the rule. The formula ϕ is its body.

A ruleset $\mathcal{R}$ is a set of rules where no two rules define the same head. For a ruleset $\mathcal{R}$ we denote $Def(\mathcal{R})$ as the set of all heads. We denote $Open(\mathcal{R})$ as all other atoms in $\mathcal{R}$.

For sets of rules where multiple rules define the same head, Hou [Hou10] defines a transformation to a ruleset: "An arbitrary set of rules [...] can be transformed into an equivalent set of rules such that each defined predicate [...] is defined by exactly one rule, [...]". A ruleset is in definition normal form (*DefNF*) iff every rule body is either a conjunction or a disjunction of atoms. By introducing new rules defining fresh atoms, any complex formula can be reduced to conjunctions and disjunctions. As all bodies are now either disjunctive or conjunctive, we call a defined atom disjunctive if its body is disjunctive and conjunctive otherwise. Thus for the rest of this chapter we only consider definitions in *DefNF*.

Definition 18 (Propositional fixpoint definitions [HDD10])**.** *Propositional fixpoint definitions are defined by simultaneous induction of least- and greatest-fixpoint definitions. A propositional least-fixpoint definition Δ is a finite expression*

$$\lfloor \mathcal{R}, d_1, \ldots, d_n \rfloor,$$

a propositional greatest-fixpoint definition ∇ is a finite expression

$$\lceil \mathcal{R}, d_1, \ldots, d_n \rceil,$$

where $\mathcal{R}$ is a ruleset, $d_1, \ldots, d_n$ are propositional least- or greatest- fixpoint definitions.

As before, for a NFD D the set of propositional atoms $Def(D)$ is the set of defined heads of all rules in D, while the set of opens, $Open(D)$, denotes all

other atoms used in bodies:

$$\begin{aligned} Def(D) &= Def(\mathcal{R}) \cup (\bigcup_i Def(d_i)) \\ Open(D) &= (Open(\mathcal{R}) \cup \bigcup_i Open(d_i)) \setminus Def(D) \end{aligned}$$

Propositional fixpoint definitions are well-formed when satisfying two constraints: (i) no two rules in the definition and sub-definitions define the same head, (ii) an atom defined in one sub-definition is not an open in another.

For a definition $D = \Delta$ or $D = \nabla$, we abuse notation to treat the definition as a set of rules.

6.2.2 Semantics

For a ruleset $\mathcal{R}$ and fitting interpretation I we define the one-step interpretation $S^{\mathcal{R}}(I)$ over $Def(\mathcal{R})$ such that the interpretation of a defined atom q is the evaluation of its rule-body ϕ_q in I:

$$S^{\mathcal{R}}(I)(q) = \phi_q^I \text{ for } (q \leftarrow \phi_q) \in \mathcal{R}$$

Since positive formulae are monotonic, $S^{\mathcal{R}}$ is monotonic: $I \leq I'$ implies $S^{\mathcal{R}}(I) \leq S^{\mathcal{R}}(I')$. This property is crucial for the calculation of fixpoints: $S^{\mathcal{R}}$ has a least- and greatest-fixpoint: $lfp(S^{\mathcal{R}})$ and $gfp(S^{\mathcal{R}})$. These fixpoints form the base-case for evaluating a fixpoint definition without sub-definitions:

Definition 19 (One-step evaluation of a fixpoint definition [HDD10])**.** *For a fixpoint definition D, with ruleset $\mathcal{R}$ and sub-definitions $d_1, \ldots d_n$, and an interpretation I for the opens of D, we define the operator $\Gamma_I^D : 2^\Sigma \to 2^\Sigma$ by mutual induction: The operator Γ_I^D returns, given an interpretation J over $Def(D)$, an interpretation $\Gamma_I^D(J) = R + E_1 + \cdots + E_n$ with:*

- *E_i is the evaluation of d_i for $(I + J)|_{Open(d_i)}$*
- *R is the one-step $S^{\mathcal{R}}(I + J|_{Def(\mathcal{R})} + E_1 + \cdots + E_n)$.*

Where the evaluation of a least fixpoint definition Δ for interpretation I is $lfp(\Gamma_I^\Delta)$. And the evaluation of a greatest fixpoint definition ∇ for interpretation I is $gfp(\Gamma_I^\nabla)$.

Theorem 6.1. *The evaluation of a fixpoint definition is well-defined.*

For the rest of the chapter we will use $\mathcal{D}$ to denote a nested fixpoint definition to avoid confusion with the edge-relation of a justification. We also assume all nested fixpoint definitions $\mathcal{D}$ are in *DefNF* over a vocabulary Σ.

6.2.3 Witness justification

Hou and Denecker [HD09] define a notion of justification for a nested fixpoint definition similar to those defined in Chapter 3. In this case, a justification serves as a witness for the evaluation of an NFD.

An atom in Σ is called a positive literal; while the set $\neg\Sigma$ denotes the set of negative literals $\{\neg q | q \in \Sigma\}$. The set $\widehat{\Sigma} = \Sigma \cup \neg\Sigma$ denotes the set of positive and negative literals. The sign of a literal l is positive if $l \in \Sigma$ and negative otherwise. We partition the set of literals of the defined atoms of $\widehat{Def(\mathcal{D})}$ in conjunctive literals, $\mathbf{C}(\mathcal{D})$, and disjunctive literals, $\mathbf{D}(\mathcal{D})$ where

$$\begin{aligned} \mathbf{C}(\mathcal{D}) &= \{q | q \text{ is conjunctive}\} \cup \{\neg q | q \text{ is disjunctive}\}, \\ \mathbf{D}(\mathcal{D}) &= \{q | q \text{ is disjunctive}\} \cup \{\neg q | q \text{ is conjunctive}\}. \end{aligned}$$

The following two definitions define a justification as a graph consisting of direct justifications. Next, we define when a justification is a witness for the evaluation of a definition.

Definition 20 (Definition graph)**.** *The definition graph $DG_{\mathcal{D}}$ of a definition $\mathcal{D}$ is the graph (Σ, E) where $E = \{(q, l) \mid (q \leftarrow \phi) \in \mathcal{D}, l \in \phi\}$. The complemented definition graph $\widehat{DG_{\mathcal{D}}}$ is the graph $(\widehat{\Sigma}, \widehat{E})$ where $\widehat{E} = E \cup \neg E$ and $\neg E = \{(\neg q, \neg l) | (q, l) \in E)\}$.*

Note that open literals of an NFD have no outgoing edges in the definition graphs. For the rest of this chapter we use the notation $G(v)$ for the set of nodes in which outgoing edges from v end. E.g., $\widehat{DG_{\mathcal{D}}}(q) = \{l \mid (q, l) \in \widehat{DG_{\mathcal{D}}}\}$.

Definition 21 (Direct justification)**.** *A set of literals DJ_q is a* direct justification *for a defined literal $q \in \widehat{Def(\mathcal{D})}$ when either $q \in \mathbf{C}(\mathcal{D})$ and $DJ_q = \widehat{DG_{\mathcal{D}}}(q)$, or $q \in \mathbf{D}(\mathcal{D})$ and DJ_q is a singleton $\{l\}$ with $l \in \widehat{DG_{\mathcal{D}}}(q)$.*

Definition 22 (Justification)**.** *A justification $\mathcal{J}$ for $\mathcal{D}$ is a directed sub-graph $(\widehat{\Sigma}, E)$ of $\widehat{DG_{\mathcal{D}}}$ such that for every literal q in $\widehat{Def(\mathcal{D})} \subseteq \widehat{\Sigma}$ the set $\mathcal{J}(q)$ is some direct justification DJ_q for q.*

Definition 23 (Supported justification)**.** *A justification $\mathcal{J}$ for $\mathcal{D}$ is supported by an interpretation $\mathcal{I}$ for Σ if for each literal l in $\widehat{Def(\mathcal{D})}$ the truth value $l^{\mathcal{I}}$ equals $(\bigwedge \mathcal{J}(l))^{\mathcal{I}}$.*

Definition 24 (Dangerous loop)**.** *A loop L in a justification $\mathcal{J}$ is* local *to a definition $\mathcal{D}$ iff all literals in L are in $Def(\mathcal{D})$ and some literal $l \in L$ is a head in the ruleset $\mathcal{R}_{\mathcal{D}}$ of $\mathcal{D}$.*

A least-loop *is a loop local to a least fixpoint definition* Δ*. A* greatest-loop *is a loop local to a greatest fixpoint definition* ∇*.*

A loop is positive *if all literals of the local fixpoint definition* $\mathcal{D}$ *are positive,* negative *if all literals are negative and* mixed *otherwise.*

A loop L *in a justification* $\mathcal{J}$ *is* dangerous *if it is either a mixed loop, a positive least-loop or a negative greatest-loop.*

As defined, mixed loops are impossible in justifications: every mixed loop has a mixed edge, namely, an edge $(p, \neg q)$ for $p, q \in \Sigma$. However, a justification is a sub-graph of the completed definition graph which has, by definition, no mixed edges. Furthermore all literals in a loop L of a justification $\mathcal{J}$ supported by an interpretation $\mathcal{I}$ have the same evaluation: if two literals in a loop have a different evaluation, there exists an edge (l, q) in this loop for which $l^{\mathcal{I}} = \mathbf{t}$ and $q^{\mathcal{I}} = \mathbf{f}$. However, since $\mathcal{J}$ is a justification, $q \in \mathcal{J}(l)$, and since $\mathcal{J}$ is supported by $\mathcal{I}$, all $q' \in \mathcal{J}(l)$ have $q'^{\mathcal{I}} = \mathbf{t}$.

Definition 25 (Witness justification)**.** *A loop* L *in a justification* $\mathcal{J}$ *is* true *for* I *if some literal in the loop is true for* I*. A loop* L *in a justification* $\mathcal{J}$ *is* safe *for* I *if it is neither true nor dangerous. A justification is a witness for a definition* $\mathcal{D}$ *and interpretation* I *if* $\mathcal{J}$ *supports* I *for* $\mathcal{D}$ *and all loops in* $\mathcal{J}$ *are safe for* I*.*

Note, if a loop is true in a supported justification then all nodes in the loop are true.

We finish this section with an example showing a witness justification of the evaluation of a definition:

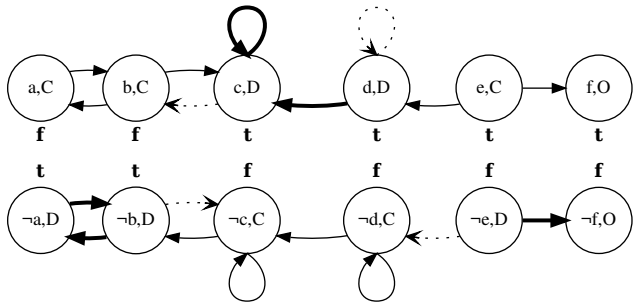


Figure 6.1: A corresponding witness justification for the nested definition in Example 9. Every literal is annotated with its type ('C' for conjunctive nodes, 'D' for disjunctive, 'O' for open) and with its evaluation (**t** and **f**). Direct justifications of conjunctive literals are solid while disjunctive literals use bold edges. Dotted edges are edges in the definition graph but not in this justification.

Example 9. Assume the following nested fixpoint definition $\mathcal{D}$ in *DefNF*:

$$\mathcal{D} = \left[\begin{array}{l} (e \leftarrow f \wedge d) \\ \left|\begin{array}{l} (b \leftarrow a \wedge c) \\ \left|\begin{array}{l} (d \leftarrow d \vee c) \\ \lceil (c \leftarrow c \vee b) \rceil \end{array}\right| \\ \lceil (a \leftarrow b \wedge b) \rceil \end{array}\right| \end{array}\right] \tag{6.1}$$

This is a nested definition over a vocabulary $\{a, b, c, d, e, f\}$ with four sub-definitions and five rules. The defined atoms $Def(\mathcal{D})$ are $a, \ldots, e$ while f is an open belonging to $Open(\mathcal{D})$. The conjunctive literals are $\mathbf{C}(\mathcal{D}) = \{a, b, e\}$ and the disjunctive literals are $\mathbf{D}(\mathcal{D}) = \{b, c\}$.
Assume the interpretation I over $Open(\mathcal{D})$: $I(f) = \mathbf{t}$. Then the evaluation of $\mathcal{D}$ is an interpretation $\mathcal{I}$ where $\mathcal{I}(a) = \mathcal{I}(b) = \mathbf{f}$ and $\mathcal{I}(c) = \mathcal{I}(d) = \mathcal{I}(e) = \mathbf{t}$.
Figure 6.1 shows a witness justification for this interpretation. The represented graph is a justification as every defined literal has a correct direct justification. This justification also supports $\mathcal{I}$. Of note is the support of conjunctive false literals b and $\neg c$ which point to both true and false literals. In contrast, the support of true literal e points only to true literals. This justification is also a witness for $\mathcal{I}$. Verify that the true loops $(\neg a, \neg b, \neg a)$ and (c, c) are not dangerous: The former is local to the definition of b, thus it is a negative least loop. The latter is a positive greatest loop local to the definition of c. Verify that the remaining loops (a, b, a), $(\neg c, \neg c)$, and $(\neg d, \neg d)$ are, although dangerous, false.

6.3 Solving fixpoint definitions by transformation

This section shows a solution method for the evaluation of a nested fixpoint definition and finding a witness via a transformation to parity games. The first subsection shows a transformation to a parametrised parity game. The second transforms the solution of such parity game back to a witness justification. The combination of both transformation coupled with a parity game solver results in a procedure to find witnessing justifications.

6.3.1 Transforming a nested fixpoint definition

In this section we define transformations between the domain of nested fixpoint definitions and that of parity games. The transformation is a simplification of the transformation for validating a μ-formula for a state machine to a parity game [EJS93; Sti95].

First, the closed definition graph $\overline{DG}_{\mathcal{D}}$ of a nested fixpoint definition $\mathcal{D}$ is defined as the graph $(\Sigma, \overline{E})$ where $\overline{E} = E_{DG_{\mathcal{D}}} \cup \{(o, o) | o \in \mathit{Open}(\mathcal{D})\}$. Then, a nested fixpoint definition in *DefNF* can be transformed to a parity game as follows:

Definition 26. *Given an NFD $\mathcal{D}$ over Σ.*

Let $(\Sigma, \overline{E})$ be the closed definition graph of $\mathcal{D}$.

Let O be the owner function determined by $\mathcal{D}$ where $O(q)$ is Even if $q \in \mathbf{C}(\mathcal{D})$, and otherwise $O(q)$ is Odd (when $q \in \mathbf{D}(\mathcal{D})$ or $q \in \mathit{Open}(\mathcal{D})$).

Let the priority key of a definition $\mathcal{D}$ be a function K mapping the definition and all its sub-definitions to $\mathbb{N}$ such that $K(\mathcal{D}_i)$ is even iff D_i is a least-fixpoint definition, and if $\mathcal{D}_i$ is a sub-definition of $\mathcal{D}_j$ then $K(\mathcal{D}_i) < K(\mathcal{D}_j)$.

Let Pr be the priority function mapping a defined atom q to the priority key $K(\mathcal{D}_i)$ of its local definition $\mathcal{D}_i$ and opens to $K(\mathcal{D})$.

Then $\mathbf{PG}_K(\mathcal{D}) = (\Sigma, \overline{E}, O, Pr)$ is a parity game derived from $\mathcal{D}$.

The owners and priorities of open nodes are, for now, chosen arbitrary. The owner of a node with a single outgoing edge is irrelevant; the priorities of opens will be overridden when calculating the evaluation for a specific initial interpretation. Additionally, since there always exists a priority key, and since every valid priority key will transform to an equivalent parity game, we will

drop the subscript and write $\mathbf{PG}(\mathcal{D})$. Note that a definition and its transformed game have equal "size".

Example 10. We continue with the nested fixpoint definition $\mathcal{D}$ of Example 9. The previous nested definition has been annotated with a priority-key:

$$\left\lceil \begin{array}{l} (e \leftarrow f \wedge d) \\ \left\lfloor \begin{array}{l} (b \leftarrow a \wedge c) \\ \left\lfloor \begin{array}{l} (d \leftarrow d \vee c) \\ \lceil (c \leftarrow c \vee b) \rceil_1 \end{array} \right\rfloor_2 \\ \lceil (a \leftarrow b) \rceil_3 \end{array} \right\rfloor_4 \end{array} \right\rceil_5 \tag{6.2}$$

The nested definition itself is assigned priority 5: odd as it is a greatest fixpoint and larger than all others.
Figure 6.2 shows the parity game $\mathbf{PG}(\mathcal{D})$ derived from this definition. Every atom is a node in the game. A head is owned by Even (square nodes) iff the rule body is conjunctive. The owner of the open f is Odd; its priority is 5. Edges go from heads to atoms in the body of the head. For example, the edge (e, d) originates from the rule $(e \leftarrow f \wedge d)$). Finally, the priority-key of the definition determines the priorities of the nodes: nodes in least-fixpoint definitions, such as the main definition, are Odd, and nodes in greatest-fixpoint definitions are Even.

Next, the initial interpretation I is considered. There are two options: First, I is transformed to a parametrisation P_I where P_I maps open literals l to Even if $I(l) = \mathbf{f}$ and to Odd if $I(l) = \mathbf{t}$. This parametrisation is then used to construct a parametrised parity game $\mathbf{PG}(\mathcal{D})_{P_I}$. Second, the previous parametrisation is embedded, resulting in the parity game $\mathbf{PG}(\mathcal{D})^{P_I}$. Recall from Chapter 2 that embedding a game with a parametrisation P_I overrides all outgoing edges of the parameters, the open literals, with simple loops (p, p) and adjusts the priorities of the open atoms to 0 if $I(l) = \mathbf{f}$ and to 1 if $I(l) = \mathbf{t}$. For convenience, we use $\mathbf{PG}(\mathcal{D}, I)$ instead of $\mathbf{PG}(\mathcal{D})^{P_I}$ to denote the parity game derived from $\mathcal{D}$ in which the interpretation I is embedded.

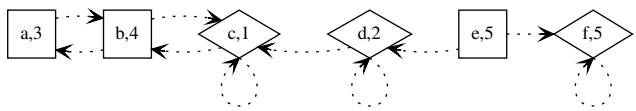


Figure 6.2: A parity game with nodes $a, \ldots, f$. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority.

Note that the interpretation can be overridden multiple times. Small changes to an interpretation will result in small changes to the transformed parity game. Section 3.6 of the third chapter shows how to maintain a safe justification while changing a parity game. Chapter 4 shows different algorithms that can solve a parity game starting from any safe justification.

6.3.2 Transforming parity game solutions to justification witnesses

Next, we define a transformation from a parity game solution back to a witness justification:

Definition 27. *Given a nested definition $\mathcal{D}$ and an interpretation I over Open($\mathcal{D}$), we define a transformation of a justification $J = (\Sigma, D, H)$ for the parity game $\mathbf{PG}(\mathcal{D}, I)$ to an interpretation $\mathcal{I}_J$ and a graph $\mathcal{J}_J = (\widehat{\Sigma}, \widehat{E})$ where*

$$\mathcal{I}_J(v) = \begin{cases} \mathbf{t} & \text{if } H(v) = Odd \\ \mathbf{f} & \text{if } H(v) = Even \end{cases}$$

$$\widehat{E} = \left\{(v,w) \middle| v \in \mathbf{D}_{\mathcal{D}}, v^{\mathcal{I}} = \mathbf{t}, w \in \widehat{DG}_{\mathcal{D}}(v), (|v|, |w|) \in J\right\} \quad (1)$$

$$\cup \left\{(v,w) \middle| v \in \mathbf{C}_{\mathcal{D}}, v^{\mathcal{I}} = \mathbf{t}, w \in \widehat{DG}_{\mathcal{D}}(v)\right\} \quad (2)$$

$$\cup \left\{(v,w) \middle| v \in \mathbf{C}_{\mathcal{D}}, v^{\mathcal{I}} = \mathbf{f}, w \in \widehat{DG}_{\mathcal{D}}(v)\right\} \quad (3)$$

$$\cup \left\{(v,w) \middle| v \in \mathbf{D}_{\mathcal{D}}, v^{\mathcal{I}} = \mathbf{f}, \mathbf{choose}\ w \in \widehat{DG}_{\mathcal{D}}(v)\right\} \quad (4)$$

Note that the justifications used for false literals v are not important as long as they are structurally correct and supported. This is why for a disjunctive false literal an arbitrary justification is chosen.

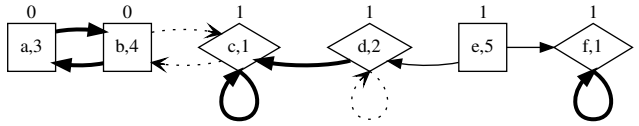


Figure 6.3: The parity game of Figure 6.2 modified for $I_O(f) = \mathbf{t}$. Square nodes are owned by Even, diamonds by Odd. The labels inside a node are the name and priority. Node f is won by Even.

Example 11. Given the NFD $\mathcal{D}$ and interpretation I of Example 9, and its transformation $\mathbf{PG}(\mathcal{D})$ as shown in Figure 6.2. Recall I is the interpretation over $\mathit{Open}(\mathcal{D})$ with $I(f) = \mathbf{t}$. The parametrisation P_I has domain $\{f\}$ and maps f to Odd. We modify the parity game $\mathbf{PG}(\mathcal{D})$ by assigning priority 1 for f. Figure 6.3 shows the modified parity game $\mathbf{PG}(\mathcal{D}, I)$ and its solution J. Nodes a and b are won by Even, nodes c, d, e and f are won by Odd.
Using Definition 27, the justification J is transformed back to an evaluation $\mathcal{I}$ where $\mathcal{I}(a) = \mathcal{I}(b) = \mathbf{f}$ and $\mathcal{I}(c) = \mathcal{I}(d) = \mathcal{I}(e) = \mathbf{t}$ and to a justification graph $\mathcal{J}$ as shown in Figure 6.1. The following lines of Definition 27 contribute the edges of the specified nodes: (1) adds the edges of nodes $\neg a, \neg b, c$, and d, (2) adds the edges of node e, (3) adds the edges of nodes $a, b, \neg c$ and $\neg d$, and (4) chose the edge $(\neg e, \neg f)$ for node $\neg e$. Another option for $\neg e$ would be the edge $(\neg e, \neg d)$.
As stated in Example 9, $\mathcal{J}$ is a justification that witnesses $\mathcal{I}$.

Lemma 6.2 (The graph $\mathcal{J}$ is a justification for $\mathcal{D}$)**.** *Given a definition $\mathcal{D}$, interpretation I for $\mathit{Open}(\mathcal{D})$, and a winning complete justification J for $\mathbf{PG}(\mathcal{D}, I)$, the graph $\mathcal{J}_J$ is a justification for $\mathcal{D}$.*

Proof. All open literals $o \in \widehat{\mathit{Open}(\mathcal{D})}$ have no justification $\mathcal{J}_J(o)$ since $\mathbf{C}$ and $\mathbf{D}$ are a partition of $\widehat{\mathit{Def}(\mathcal{D})}$. For all conjunctive literals q, the set $\mathcal{J}_J(q) = DG_{\mathcal{D}}(q)$ is a valid direct justification by definition. For all false disjunctive literals an arbitrary direct justification is chosen. The direct justifications for true disjunctive literals v remain. Clearly $\mathcal{J}_J(v)$ is a subset of $\widehat{DG}_{\mathcal{D}}(v)$, hence we must prove $\mathcal{J}_J(v)$ is a singleton. Two cases are considered, based on the sign of the literal:

i) A true disjunctive positive literal q has $q^{\mathcal{I}} = \mathcal{I}(q) = \mathbf{t}$, and belongs to both $\mathbf{D}(\mathcal{D})$ and Σ. By definition of $\mathcal{I}$, $H(q) = \mathit{Odd} = O(q)$, so J has, for q, a single outgoing edge (q, l) which, by definition of $\mathbf{PG}(\mathcal{D}, I)$, also belongs to $DG_{\mathcal{D}}$. Thus $\mathcal{J}_J(q) = \{l\}$ since $(|q|, |l|) = (q, l) \in J$, and since no other literal l' in $\widehat{DG}_{\mathcal{D}}(q)$ has $|l'| = l$.

ii) A true disjunctive negative literal $\neg q$ has $(\neg q)^{\mathcal{I}} = \mathbf{t}$, belongs to both $\mathbf{D}(\mathcal{D})$ and $\neg\Sigma$. Its complement, q, has $\mathcal{I}(q) = \mathbf{f}$ and is conjunctive. By Definition 26, $O(q)$ is *Even*. By definition of $\mathcal{I}$, $H(q) = Even = O(q)$, so J has, for q, a single outgoing edge (q, l). This edge proves l is in $DG_{\mathcal{D}}(q)$, and we know $\neg l$ is in $\widehat{DG}_{\mathcal{D}}(\neg q) = \{\neg r | r \in DG_{\mathcal{D}}(q)\}$. Thus $\neg l \in \mathcal{J}_J(\neg q)$ since $\neg l \in \widehat{DG}_{\mathcal{D}}(\neg q)$ and $(|\neg q|, |\neg l|) = (q, l) \in J$. Finally, no other literal w in $\widehat{DG}_{\mathcal{D}}(\neg q)$ exists such that $|w| = l$ thus $\mathcal{J}_J(\neg q)$ is the singleton $\{\neg l\}$.

In conclusion, for all defined literals l is $\mathcal{J}(l)$ a direct justification for l. □

Lemma 6.3 ($\mathcal{I}$ supports $\mathcal{J}$ for D)**.** *Given a definition D, interpretation I for Open(D), and a winning complete justification justification J for $\mathbf{PG}(\mathcal{D}, I)$, the justification $\mathcal{J}_J$ for $\mathcal{D}$ is supported by $\mathcal{I}_J$.*

Proof. We prove all literals $l \in \widehat{Def(d)}$ have $l^{\mathcal{I}} = (\bigwedge \mathcal{J}(l))^{\mathcal{I}}$:

If l is a true disjunctive literal then $\mathcal{J}(l)$ is a singleton $\{l'\}$ and $(|l|, |l'|)$ is in J. Since J is weakly winning, $H(|l|) = H(|l'|)$. Furthermore, since l and l' have the same sign (both are positive or both are negative), $l^{\mathcal{I}}_J = l'^{\mathcal{I}}_J$.

If l is a true conjunctive literal then $\mathcal{J}(l)$ is $\widehat{DG}_{\mathcal{D}}(l)$. If l is positive then $H(l) = Even \neq O(l) = Odd$, thus $(l, l') \in J$ for all $l' \in \widehat{DG}_{\mathcal{D}}(l)$. Since J is weakly-winning, $H(l) = H(l')$, and so $l^{\mathcal{I}} = l'^{\mathcal{I}}$.

If l is a false disjunctive literal then any w in $\widehat{DG}_{\mathcal{D}}(l)$ is false as well: The negation of l is a true conjunctive literal whose direct justification is $\widehat{DG}_{\mathcal{D}}(\neg l)$. Thus $l^{\mathcal{I}} = (\bigwedge \mathcal{J}(l))^{\mathcal{I}}$ for all possibly chosen w from $\widehat{DG}_{\mathcal{D}}(l)$.

If l is a false conjunctive literal then one of the w in $\widehat{DG}_{\mathcal{D}}(l)$ is false: the negation of l is a true disjunctive literal which has a singleton direct justification $\{l'\}$. Thus $l^{\mathcal{I}} = (\bigwedge \mathcal{J}(l))^{\mathcal{I}} = (\bigwedge \widehat{DG}_{\mathcal{D}}(l))^{\mathcal{I}} = \mathbf{f}$ since $\neg l'$ is in $\widehat{DG}_{\mathcal{D}}(l)$. □

Theorem 6.4 ($\mathcal{J}$ is a witness for D)**.** *Given a definition D, interpretation I for Open(D), and a winning complete justification justification J for $\mathbf{PG}(\mathcal{D}, I)$, the justification $\mathcal{J}_J$ for $\mathcal{D}$ witnesses $\mathcal{I}_J$.*

Proof. Lemma 6.3 proves $\mathcal{J}_J$ is supported by $\mathcal{I}_J$, thus it remains to prove every loop in $\mathcal{J}_J$ is not dangerous. Take any true loop $v_1, v_2, \ldots, v_n, v_1$ in $\mathcal{J}_J$ local to a (sub-)definition D'. First we prove $|v_1|, |v_2|, , \ldots, |v_n|, |v_1|$ is a loop in the justification J by distinguishing three cases: i) if v_i is a disjunctive true literal then $(|v_i|, |v_{i+1}|) \in J$ by Definition 27. ii) If v_i is a conjunctive positive literal then v_i is a node in the solved parity game. The body of v_i is a conjunction thus

every v_i has an edge to every literal in the body and v_i is owned by Even. Since v_i^I is true, $H(v_i) = Odd$. As $O(v_i)$ loses v_i, all edges including (v_i, v_{i+1}) are in the complete justification J. iii) If v_i is a conjunctive negative literal then $|v_i|$ is a node in the game owned by Odd (since the body of $|v_i|$ is a disjunction). Since v_i is true, $H(|v_i|) = Even \neq O(|v_i|)$. As $O(v_i)$ loses v_i, the edge $(|v_i|, |v_{i+1}|)$ belongs to J.

Next we prove the highest priority of this loop is $K(D')$. This loop has a highest priority $p = max\{Pr(v_1), \ldots, Pr(v_n)\}$. Every literal v_i in the true loop is defined by an atom in a sub-definition D'' of D'. Thus $Pr(|v_i|) = K(D'') \leq K(D')$ and at least one v_i belongs to D'. It follows that $p = K(D')$. A simple case distinction remains. If the loop is positive then all $|v_i|$ are won by Odd. Thus $p = K(D')$ is odd and D' is a least fixpoint. On the contrary, if the loop is negative, all $|v_i|$ are won by Even. Thus $K(D')$ is Even and D' is a greatest fixpoint. In either case the true loop is not dangerous. □

Finally, all transformations are tied together in a procedure, *FindWitness*. Theorem 6.4 proves the ability of *FindWitness* for finding models for NFD's.

Proc *FindWitness*$(\mathcal{D}, I)$**:**
  $\mathcal{PG} \leftarrow \mathbf{PG}(\mathcal{D}, I)$
  Find a justification J which solves the game $\mathcal{PG}$
  Construct interpretation $\mathcal{I}_J$ and justification $\mathcal{J}_J$ from $\mathcal{D}$ and J.
  return $\mathcal{I}_J, \mathcal{J}_J$

6.4 Solving parity games by transformation

The opposite transformation exists as well: we show parity games can be solved by finding a witness for a transformed NFD. Moreover, when considering the same priority key, the opposite transformation is the right inverse of $\mathbf{PG}(\mathcal{D})$.

6.4.1 Transforming PPG's to nested fixpoint definitions

This section defines a transformation from parity games to nested fixpoint definitions. It is roughly the inverse of the transformation defined in Definition 26. However, the derived parity game lacks the nesting structure of a definition. The reverse transformation cannot reconstruct the original definition. Instead, an equivalent definition is constructed which has, in particular, a linear nesting structure.

Definition 28. *Let $(V, E, O, Pr)_P$ be a parametrised parity game to transform to a nested fixpoint definition with opens P_{01} and interpretation I_P. The propositional atoms of the definition are the set of nodes V. First, all nodes $v \notin P$ with outgoing edges $E(v)$ and owned by α is transformed in a rule $(v \leftarrow \bigoplus_{w \in E(v)} w)$ where, if α is Even, $\oplus = \wedge$ and otherwise $\oplus = \vee$.*

Next, the rules are partitioned in rulesets according to their priority: Order the priorities in $dom(Pr)$ in a sequence $p_1 < p_2 < \cdots < p_n$. Then the k-th ruleset is $\mathcal{R}_k = \{(v \leftarrow \phi_v) | v \in V, Pr(v) = p_k\}$.

Finally, the rulesets are organised in a nested fixpoint definition: Let $D_1 = \mathcal{R}_1$, let D_k for $k = 2 \ldots n$ be $\lfloor \mathcal{R}_k, D_{k-1} \rfloor$ if p_k is Even and $\lceil \mathcal{R}_k, D_{k-1} \rceil$ if p_k is Odd. Then the transformed nested fixpoint definition, $\mathbf{NFD}(\mathcal{PG}_P)$, is D_n, and the interpretation I_P is an interpretation mapping $v \in P_{01}$ to $\mathbf{t}$ if $P(v) = Odd$ and $\mathbf{f}$ otherwise. Optionally this transformation produces a priority key $K_{\mathcal{PG}_P}$ which maps D_k to p_k.

Again the transformation conserves the "size". Every node corresponds to an atom. Every edge corresponds to an atom in a body. And, every priority corresponds to a nested definition. As stated before, transforming a parity game to a nested fixpoint definition is roughly the inverse of the previous transformation:

Theorem 6.5. *Let $\mathcal{PG}_P$ be a parametrised parity game, let $\mathcal{D}$ be the transformed NFD, $\mathbf{NFD}(\mathcal{PG}_P)$, let I_P be the open-interpretation determined by P, and let K be the optionally produced priority key, then $\mathbf{PG}_K(\mathcal{D}, I_P) = \mathcal{PG}^P$.*

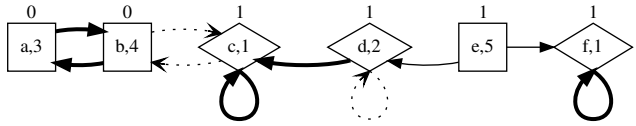


Figure 6.4: The transformed parity game of the running example with a winning justification, a copy of Figure 6.3

Example 12. Figure 6.4 shows a parity game to be transformed to an NFD. First, the priorities are sorted. The priorities are consecutive, so $p_k = k$. Next, we build the rulesets:

$$\begin{aligned} R_1 &= \{(c \leftarrow c \vee b), (f \leftarrow f)\} \\ R_2 &= \{(d \leftarrow c \vee d)\} \\ R_3 &= \{(a \leftarrow b)\} \\ R_4 &= \{(b \leftarrow a \wedge c)\} \\ R_5 &= \{(e \leftarrow d \wedge f)\} \end{aligned}$$

Finally we nest the rulesets. The most nested definition is $\mathcal{D}_1 = \lceil R_1 \rceil = \lceil (c \leftarrow c \vee b), (f \leftarrow f) \rceil$. The full definition is then:

$$\mathcal{D}_5 = \left\lceil \begin{array}{l} (e \leftarrow f \wedge d) \\ \left\lfloor \begin{array}{l} (b \leftarrow a \wedge c) \\ \left\lceil \begin{array}{l} (a \leftarrow b) \\ \left\lfloor \begin{array}{l} (d \leftarrow d \vee c) \\ \left\lceil \begin{array}{l} (c \leftarrow c \vee b) \\ (f \leftarrow f) \end{array} \right\rceil \end{array} \right\rfloor \end{array} \right\rceil \end{array} \right\rfloor \end{array} \right\rceil \tag{6.3}$$

Note the similarity of this definition to Definition 6.1 in the first example: removing the rule $(f \leftarrow f)$ results in an equivalent definition.

6.4.2 Transforming witness justifications to PPG solutions

While the back-transformation and Theorem 6.4 shows some witnesses are related to solutions it remains to be proven that all witnesses can be transformed to a solution for the original parity game. The transformation is the inverse of Definition 27.

Definition 29. *An interpretation $\mathcal{I}$ and witness $\mathcal{J} = (\widehat{V}, \widehat{E})$ can be transformed to a parity game justification $J_{\mathcal{J}} = (V, D, H)$ where*

$$\begin{aligned} H(v) &= \begin{cases} Odd & \text{if } \mathcal{I}(v) = \mathbf{t} \\ Even & \text{if } \mathcal{I}(v) = \mathbf{f} \end{cases} \\ D &= \left\{ (|v|, |w|) \middle| (v, w) \in \widehat{E}, v^{\mathcal{I}} = \mathbf{t} \right\} \end{aligned}$$

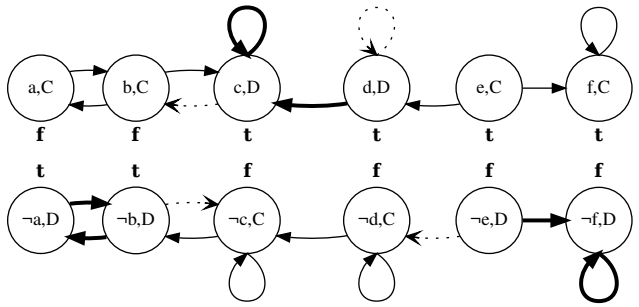


Figure 6.5: A corresponding witness justification for the nested definition in Example 12. Every literal is annotated with its type ('C' for conjunctive nodes, 'D' for disjunctive) and with its evaluation (**t** and **f**). Direct justifications of conjunctive literals are solid while disjunctive literals use bold edges. Dotted edges are unused.

Example 13. We continue Example 12: The full definition, $\mathcal{D}_5$, has no opens. The evaluation of $\mathcal{D}_5$ is an interpretation $\mathcal{I}$ with $\mathcal{I}(a) = \mathcal{I}(b) = \mathbf{f}$ and $\mathcal{I}(c) = \mathcal{I}(d) = \mathcal{I}(e) = \mathcal{I}(f) = \mathbf{t}$. Figure 6.5 shows a witnessing justification for $\mathcal{I}$. Definition 29 says this interpretation and witness justification can be transformed to a parity game justification (V, D, H) where $H(a) = H(b) = \mathit{Even}$, $H(c) = H(d) = H(e) = H(f) = \mathit{Odd}$, and graph D:

$$D = \{(a, b), (b, a)(c, c), (d, c), (e, d), (e, f), (f, f)\}$$

This is exactly the justification shown in Figure 6.4.

The previous example shows we can solve parity games by transforming them to a nested fixpoint definition and solving it instead.

Proc *FindJustification*($\mathcal{PG}$)**:**
  Transform $\mathcal{PG}$ to a nested fixpoint definition $\mathcal{D}$
  Find the interpretation $\mathcal{I}$ and a witness $\mathcal{J}$ for $\mathcal{D}$
  Transform$(\mathcal{I}, \mathcal{J})$ to a justification J
  return J

Theorem 6.6. *Let $\mathcal{PG}$ be a parity game, let $P_\emptyset$ be the trivial empty parameter function, let $\mathcal{D}$ be the transformed NFD, $\mathbf{NFD}(\mathcal{PG}_{P_\emptyset})$, let the interpretation $\mathcal{I}$ over Open($\mathcal{D}$) and given a justification $\mathcal{J}$ for $\mathcal{D}$ be such that $\mathcal{J}$ is a witness for $\mathcal{I}$, then $J_\mathcal{J}$ is a winning justification for $\mathcal{PG}$.*

The proof is analogous to those of Theorems 6.3 and 6.4. Thus only a sketch of the proof is provided.

Let $\mathcal{PG}$ be a parity game and let $\mathcal{D}, \mathcal{I}, \mathcal{J}, J$ be as defined in Theorem 6.6. First, to prove J is weakly winning, take any edge (v, w) in J. Then either (v, w) or $(\neg v, \neg w)$ is an edge in $\mathcal{J}$. Since $\mathcal{J}$ supports $\mathcal{I}$, $v^{\mathcal{I}} = w^{\mathcal{I}}$, and so $H(v) = H(w)$. Second, to prove J is winning, take any play $\langle v_1, v_2, \dots \rangle$ and prove the winner is $H(v_1)$. This play has a highest priority p and a winner $\alpha = p \mod 2$. Assume α is Odd. Any sequence of nodes in the play starting and ending in the same node v_i with priority p will be a true loop in $\mathcal{J}$ local to a least fixpoint definition. This least fixpoint definition links the priority p to the interpretation of v_i to prove $H(v_i) = H(v_1) = \alpha$. When α is Even, every node in such a sequence must be negated to obtain a true loop in $\mathcal{J}$. This time, the true loop will be local to a greatest fixpoint definition.

6.5 Conclusion

The transformations show nested fixpoint definitions and parity games are closely related. This is further evidenced by the properties of these transformations: each has a linear time-complexity. Moreover, the transformations preserve the size: there is a one-to-one mapping between atoms and nodes and between the size of a body for a literal and the outgoing edges of the corresponding node.

As both contain unpreserved redundant information, the transformations are not bijective: in one direction, transforming a nested fixpoint definitions discards the nesting structure. To improve efficiency, some parity game solvers recover this structure with a dynamic SCC analysis [FL09a; FL09b]. The distinction between open and defined atoms is similarly discarded: a fixed interpretation is required and is encoded in the resulting game. In the other direction, transforming a parity game discards priority information, as only the parity and relative ordering is relevant. Likewise, most solvers renumber the priorities of parity games as an optimisation [FL09a].

6.5.1 Future work

This close relation indicates that, as done with justifications, more techniques, algorithms and applications can be shared between both representations. For example, we can expect that optimisations for parity games carry over to NFD's. Another idea is to define **Justify** for NFD's instead of for parity games. Then, solving an NFD is possible with all algorithms in Chapter 5. In

fact, the algorithm described in Chapter 6 of [Hou10] is similar to the nested fixpoint iteration algorithm with justifications, as Hou provided the inspiration for using justifications and both are based on the well-known nested fixpoint iteration algorithm. As such, using state-of-the-art parity game algorithms should improve the SAT(FD) solver.

Differently, while NFD's have "opens", parity games lack this concept. In SAT(FD) these opens are used to find models for an NFD. It is plausible that applications using parity games can also benefit from this idea. For example to solve the question "Is there a configuration for which the machine is unfair?". Generally, this expression is transformed to a parity game by allowing the antagonist to choose any configuration and solving fairness for all sub-games. However, a similar graph is generated for each combination of values. Approaching the problem with SAT(FD) would transform to a parametrised parity game to be solved as within SAT modulo theory: "Can we choose the winners of parameter nodes such that the fairness node is won by the antagonist?" This type of problems is then solved by the state-of-the-art of both domains: fairness is verified using parity games while SAT-solvers excel in solving configuration problems.

Chapter 7

Conclusion

As stated in the introduction, the initial goal of the research was to apply justifications to the nested fixpoint algorithm. Yet, in search of a method to validate the implementation, we noticed parity games not only lacked approaches similar to justifications but also had a wider range of applications. Furthermore, we observed that justifications are equally applicable to multiple wildly different algorithms. As a result, the main goal was adjusted to applying justifications to these algorithms and formalising this in a unifying framework.

7.1 Contributions

In this text, we showed justifications can be used as a common basis to explain multiple algorithms for solving parity-games. The preliminaries (Chapter 2) served as an introduction for understanding parity games and their primary algorithms. Also, a small extension of a parity game, a parametrised parity game, was defined.

In Chapter 3, we proposed justifications as a new alternative for solving parity games. This chapter consists of three parts: first, a justification data-structure was defined along with additional properties. E.g., when a justification is winning it provides a solution for a certain parametrised parity game. Second, an operation, **Justify**, was detailed with special invariants to ensure justifying a node in a safe justification remains safe and progresses towards a solution of the parity game. We proved repeating this operation until fixpoint eventually solves parity games. The third part prepares other ways to exploit justifications:

- Justifying multiple nodes at once leads to practical improvements in Chapter 4.
- Maintaining a safe justification while modifying a parity game unlocks new research tracks.

The main goal of this text is realised in Chapter 4. To show that **Justify** can serve as a foundation for solving parity games, we restated existing algorithms by replacing the internal state with a justification and manipulating this justification with nothing but **Justify**. In total we restated four state-of-the-art recursive parity game solving algorithms: fixpoint iteration, Zielonka's algorithm, priority promotion and tangle learning.

Chapter 5 revisits the original goal of validating the improved implementation. With the expanded scope, all justification algorithms of Chapter 4 were implemented. This chapter experimentally compares the new algorithms to the existing state-of-the-art with encouraging results. Each justification implementation improved upon the respective original. Furthermore, one family of crafted parity games, Jurdziński-type games, was identified for which the time complexity was reduced from exponential to polynomial.

Finally, Chapter 6 links back to the original inspiration. We formalised an equivalence between the justifications for parity games and the justifications for nested fixpoint definitions of Hou et al. [HDD10]. Specifically, four transformations were constructed: from nested fixpoint definitions to parity games together with a transformation from a parity game solution (a winning justification) back to a nested fixpoint solution (a witness justification), and a transformation from parity games to nested fixpoint definitions coupled with a transformation from a witness justification back to a winning justification.

7.2 Future work

Considering the justifications as defined in Chapter 3, one of the research questions is whether further reductions in the number of reset nodes are possible. The conclusion of this chapter already discusses one opportunity. Yet only an implementation and benchmark will give a definitive answer. Another variant of **Justify** try choosing a different safe direct justification in an effort to further reduce the number of reset nodes.

Another unsolved question is which order of justifications and which of the valid direct justifications is best. Chapter 4 shows that algorithms using justifications with large direct justification levels generally perform better. However, none of the algorithms tries to predict the increase when closing regions and tangles. The most similar approach is delayed priority promotion [BDM18] and the non-

on-the-fly tangle learning implementation [van18b], which try to close regions for all priorities, but only after attracting nodes to the highest possible region. Consider the following example scenario in which a simple look-ahead is superior: Assume a justification in which a node has two direct justifications with which it can be justified. The first direct justification is to a larger justification level to another node, the second is to the current justification level. Then the attraction procedure always forces the first direct justification, only fixpoint induction can, by chance, deviate. However, for this scenario[1], the second direct justification forms a dominion. Clearly, the second choice is better. Efficiently identifying such opportunities can further improve parity game solvers.

With the examples provided by the reconstructed algorithms, a big research challenge is finding which other algorithms are improved by using justifications. We identified two opportunities. First is an immediate integration of a justification with the idea that a justification will reduce the number of nodes which are reset. One excellent candidate is Parys' quasi-polynomial algorithm [Par19], whose design during solving includes frequently aborting, resetting nodes, and restarting.

Second, since many other parity-game algorithms don't reset nodes, another approach is adapting the justifications to fit these algorithms, ideally without losing its current capabilities. Prime targets include various progress measures [Jur00; Fea+17]. The assignment of a progress measure for every node is similar to the justification level of each node. With the difference that progress measures are more complex in order to increase monotonically while the justification level decreases for nodes when reset. Thus another opportunity consists of replacing the domain of the justification level with a set of progress measures.

For Chapter 5, future work is expected to focus on building a more difficult benchmark. One interesting development is the recent parity game solving track of the Reactive Synthesis Competition [JP20; JP21], which should provide a steady influx of new parity games.

Finally, in Chapter 6, the most interesting research question generated by linking parity games to nested fixpoint definitions is the possibility of SAT modulo parity games, or SAT modulo CTL*. The work by Hou et al. [HDD10] shows how such a modulo theory can work and Chapter 6 shows how this can be improved by using faster algorithms such as tangle learning. For now this is a solution for a non-existing problem. Finding applications is compounded with the fact that simpler winning conditions, such as reachability and safety, can already be modelled with SAT modulo closure or with answer set programming.

[1] A less extreme scenario consists of deriving a tangle that can be promoted to a justification level larger than the justification level of the first direct justification

Bibliography

[AS09] G. Audemard and L. Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by C. Boutilier. 2009, pp. 399–404. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

[BDM16a] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Improving Priority Promotion for Parity Games". In: *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*. Ed. by R. Bloem and E. Arbel. Vol. 10028. Lecture Notes in Computer Science. 2016, pp. 117–133. DOI: 10.1007/978-3-319-49052-6_8.

[BDM16b] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "Solving Parity Games via Priority Promotion". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 270–290. DOI: 10.1007/978-3-319-41540-6_15.

[BDM18] M. Benerecetti, D. Dell'Erba, and F. Mogavero. "A delayed promotion policy for parity games". In: *Inf. Comput.* 262 (2018), pp. 221–240. DOI: 10.1016/j.ic.2018.09.005.

[BFL14] F. Bruse, M. Falk, and M. Lange. "The Fixpoint-Iteration Algorithm for Parity Games". In: *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014*. Ed. by A. Peron and C. Piazza. Vol. 161. EPTCS. 2014, pp. 116–130. DOI: 10.4204/EPTCS.161.12.

[BL04] R. J. Brachman and H. J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004. ISBN: 978-1-55860-932-7. URL: http://www.elsevier.com/wps/find/bookdescription.cws%5C_home/702602/description.

[BSK17] R. Bloem, S. Schewe, and A. Khalimov. "CTL* synthesis via LTL synthesis". In: *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. Ed. by D. Fisman and S. Jacobs. Vol. 260. EPTCS. 2017, pp. 4–22. DOI: 10.4204/EPTCS.260.4.

[BVW94] O. Bernholtz, M. Y. Vardi, and P. Wolper. "An Automata-Theoretic Approach to Branching-Time Model Checking (Extended Abstract)". In: *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*. Ed. by D. L. Dill. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 142–155. DOI: 10.1007/3-540-58179-0_50.

[BW18] B. Bogaerts and A. Weinzierl. "Exploiting Justifications for Lazy Grounding of Answer Set Programs". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by J. Lang. ijcai.org, 2018, pp. 1737–1745. DOI: 10.24963/ijcai.2018/240.

[Cal+17] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. "Deciding parity games in quasipolynomial time". In: *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*. Ed. by H. Hatami, P. McKenzie, and V. King. ACM, 2017, pp. 252–263. DOI: 10.1145/3055399.3055409.

[CGR11] S. Cranen, J. F. Groote, and M. A. Reniers. "A linear translation from CTL* to the first-order modal μ -calculus". In: *Theor. Comput. Sci.* 412.28 (2011), pp. 3129–3139. DOI: 10.1016/j.tcs.2011.02.034.

[Cla+18] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8.

[Cra+13] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. "An Overview of the mCRL2 Toolset and Its Recent Advances". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Piterman and S. A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 199–213. ISBN: 978-3-642-36742-7. DOI: 10.1007/978-3-642-36742-7_15.

[DD93] M. Denecker and D. De Schreye. "Justification Semantics: A Unifiying Framework for the Semantics of Logic Programs". In: *Logic Programming and Non-monotonic Reasoning, Proceedings of the Second International Workshop, Lisbon, Portugal, June 1993*. Ed. by L. M. Pereira and A. Nerode. MIT Press, 1993, pp. 365–379.

[DDS12] B. De Cat, M. Denecker, and P. J. Stuckey. "Lazy Model Expansion by Incremental Grounding". In: *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*. Ed. by A. Dovier and V. S. Costa. Vol. 17. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 201–211. DOI: 10.4230/LIPIcs.ICLP.2012.201.

[DS69] J. W. De Bakker and D. Scott. *A Theory of Programs: In Outline of Joint Work*. IBM, 1969.

[DT08] M. Denecker and E. Ternovska. "A logic of nonmonotone inductive definitions". In: *ACM Trans. Comput. Log.* 9.2 (2008), 14:1–14:52. DOI: 10.1145/1342991.1342998.

[Dut14] B. Dutertre. "Yices 2.2". In: *Computer-Aided Verification (CAV'2014)*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, July 2014, pp. 737–744.

[EJ91] E. A. Emerson and C. S. Jutla. "Tree Automata, Mu-Calculus and Determinacy (Extended Abstract)". In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 368–377. DOI: 10.1109/SFCS.1991.185392.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. "On Model-Checking for Fragments of μ-Calculus". In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by C. Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 385–396. DOI: 10.1007/3-540-56922-7_32.

[EL86] E. A. Emerson and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus (Extended Abstract)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 267–278.

[Fea+17] J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. "An ordered approach to solving parity games in quasi polynomial time and quasi linear space". In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of*

Software, Santa Barbara, CA, USA, July 10-14, 2017. Ed. by H. Erdogmus and K. Havelund. ACM, 2017, pp. 112–121. DOI: 10.1145/3092282.3092286.

[Fea+19] J. Fearnley, S. Jain, B. de Keijzer, S. Schewe, F. Stephan, and D. Wojtczak. "An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space". In: *Int. J. Softw. Tools Technol. Transf.* 21.3 (2019), pp. 325–349. DOI: 10.1007/s10009-019-00509-3.

[FL09a] O. Friedmann and M. Lange. "Solving Parity Games in Practice". In: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*. Ed. by Z. Liu and A. P. Ravn. Vol. 5799. Lecture Notes in Computer Science. Springer, 2009, pp. 182–196. DOI: 10.1007/978-3-642-04761-9_15.

[FL09b] O. Friedmann and M. Lange. "The PGSolver collection of parity game solvers". In: *University of Munich* (2009).

[Gae95] B. Gaede. "Superposition extended with Sorts". PhD thesis. Universität Kaiserslautern Saarbrücken, 1995.

[GLM11] C. L. Goues, K. R. M. Leino, and M. Moskal. "The Boogie Verification Debugger (Tool Paper)". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by G. Barthe, A. Pardo, and G. Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 407–414. DOI: 10.1007/978-3-642-24690-6_28.

[GTW02] E. Grädel, W. Thomas, and T. Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-00388-6. DOI: 10.1007/3-540-36387-4.

[HD09] P. Hou and M. Denecker. "A logic of fixpoint definitions". In: *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) 2009*. 2009, pp. 1–15.

[HDD10] P. Hou, B. De Cat, and M. Denecker. "FO(FD): Extending classical logic with rule-based fixpoint definitions". In: *Theory Pract. Log. Program.* 10.4-6 (2010), pp. 581–596. DOI: 10.1017/S1471068410000293.

[HK66] A. J. Hoffman and R. M. Karp. "On nonterminating stochastic games". In: *Management Science* 12.5 (1966), pp. 359–370.

[Hou10] P. Hou. “Techniques for reasoning in FO (ID) and fixpoint logic”. PhD thesis. Ph. D. thesis, Department of Computer Science, KU Leuven, Leuven, Belgium, 2010.

[Jac+19] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker. “The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results”. In: *CoRR* abs/1904.07736 (2019). arXiv: 1904.07736. URL: http://arxiv.org/abs/1904.07736.

[Jan+16] J. Jansen, B. Bogaerts, J. Devriendt, G. Janssens, and M. Denecker. “Relevance for SAT(ID)”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*. Ed. by S. Kambhampati. IJCAI/AAAI Press, 2016, pp. 596–602. URL: http://www.ijcai.org/Abstract/16/091.

[JL17] M. Jurdziński and R. Lazic. “Succinct progress measures for solving parity games”. In: *CoRR* abs/1702.05051 (2017). arXiv: 1702.05051. URL: http://arxiv.org/abs/1702.05051.

[JP20] S. Jacobs and G. A. Pérez. *SYNTCOMP 2020 Results*. 2020. URL: http://www.syntcomp.org/syntcomp-2020-results/.

[JP21] S. Jacobs and G. A. Pérez. *SYNTCOMP 2021: Call for Benchmarks and Solvers*. 2021. URL: http://www.syntcomp.org/syntcomp-2021-call-for-benchmarks-and-solvers/.

[Jur00] M. Jurdziński. “Small Progress Measures for Solving Parity Games”. In: *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*. Ed. by H. Reichel and S. Tison. Vol. 1770. Lecture Notes in Computer Science. Springer, 2000, pp. 290–301. DOI: 10.1007/3-540-46541-3_24.

[Kei15] J. J. A. Keiren. “Benchmarks for Parity Games”. In: *Fundamentals of Software Engineering - 6th International Conference, FSEN 2015 Tehran, Iran, April 22-24, 2015, Revised Selected Papers*. Ed. by M. Dastani and M. Sirjani. Vol. 9392. Lecture Notes in Computer Science. Springer, 2015, pp. 127–142. DOI: 10.1007/978-3-319-24644-4_9.

[Kes+93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. “A Decision Algorithm for Full Propositional Temporal Logic”. In: *Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. Ed. by C. Courcoubetis.

Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 97–109. DOI: 10.1007/3-540-56922-7_9.

[KP12] G. Kant and J. van de Pol. "Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games". In: *Proceedings First Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2012, Tallinn, Estonia, 1st April 2012*. Ed. by A. Wijs, D. Bosnacki, and S. Edelkamp. Vol. 99. EPTCS. 2012, pp. 50–65. DOI: 10.4204/EPTCS.99.7.

[LBD20] R. Lapauw, M. Bruynooghe, and M. Denecker. "Improving Parity Game Solvers with Justifications". In: *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*. Ed. by D. Beyer and D. Zufferey. Vol. 11990. Lecture Notes in Computer Science. Springer, 2020, pp. 449–470. ISBN: 978-3-030-39321-2. DOI: 10.1007/978-3-030-39322-9_21.

[LBDar] R. Lapauw, M. Bruynooghe, and M. Denecker. "Justifications and Reconstructing Parity Game Algorithms". In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems, Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday* (2020 (to appear)).

[LMS20] M. Luttenberger, P. J. Meyer, and S. Sickert. "Practical synthesis of reactive systems from LTL specifications via parity games". In: *Acta Informatica* 57.1-2 (2020), pp. 3–36. DOI: 10.1007/s00236-019-00349-3.

[Lon+94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. "An Improved Algorithm for the Evaluation of Fixpoint Expressions". In: *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*. Ed. by D. L. Dill. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 338–350. DOI: 10.1007/3-540-58179-0_66.

[Mar+08] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. "SAT(ID): Satisfiability of Propositional Logic Extended with Inductive Definitions". In: *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*. Ed. by H. K. Büning and X. Zhao. Vol. 4996. Lecture Notes in Computer Science. Springer, 2008, pp. 211–224. DOI: 10.1007/978-3-540-79719-7_20.

[MB08] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MC18] T. Michaud and M. Colange. "Reactive synthesis from LTL specification with Spot". In: *Proceedings of the 7th Workshop on Synthesis, SYNT@ CAV*. 2018.

[McC10] W. McCune. "Prover9 and Mace4". 2005–2010. URL: http://www.cs.unm.edu/~mccune/prover9/.

[MSL18] P. J. Meyer, S. Sickert, and M. Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by H. Chockler and G. Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31.

[Par19] P. Parys. "Parity Games: Zielonka's Algorithm in Quasi-Polynomial Time". In: *44th International Symposium on Mathematical Foundations of Computer Science, MFCS 2019, August 26-30, 2019, Aachen, Germany*. Ed. by P. Rossmanith, P. Heggernes, and J.-P. Katoen. Vol. 138. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 10:1–10:13. DOI: 10.4230/LIPIcs.MFCS.2019.10.

[Pit06] N. Piterman. "From Nondeterministic Buchi and Streett Automata to Deterministic Parity Automata". In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 2006, pp. 255–264. DOI: 10.1109/LICS.2006.28.

[Saf88] S. Safra. "On the Complexity of omega-Automata". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948.

[Sch08] S. Schewe. "An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games". In: *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*.

Ed. by M. Kaminski and S. Martini. Vol. 5213. Lecture Notes in Computer Science. Springer, 2008, pp. 369–384. DOI: 10.1007/978-3-540-87531-4_27.

[Sei96] H. Seidl. "Fast and Simple Nested Fixpoints". In: *Inf. Process. Lett.* 59.6 (1996), pp. 303–308. DOI: 10.1016/0020-0190(96)00130-5.

[Sti95] C. Stirling. "Lokal Model Checking Games". In: *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*. Ed. by I. Lee and S. A. Smolka. Vol. 962. Lecture Notes in Computer Science. Springer, 1995, pp. 1–11. DOI: 10.1007/3-540-60218-6_1.

[Tar55] A. Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.

[Tar72] R. E. Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[Tri18] Tricentis. *Tricentis Software Fail Watch Finds 3.6 Billion People Affected and $1.7 Trillion Revenue Lost by Software Failures Last Year*. 2018. URL: https://www.tricentis.com/news/tricentis-software-fail-watch-finds-3-6-billion-people-affected-and-1-7-trillion-revenue-lost-by-software-failures-last-year.

[van18a] T. van Dijk. "Attracting Tangles to Solve Parity Games". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by H. Chockler and G. Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 198–215. DOI: 10.1007/978-3-319-96142-2_14.

[van18b] T. van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by D. Beyer and M. Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 291–308. DOI: 10.1007/978-3-319-89960-2_16.

[van19] T. van Dijk. "A Parity Game Tale of Two Counters". In: *Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2-3rd September 2019*. Ed. by J. Leroux and J.-F. Raskin. Vol. 305. EPTCS. 2019, pp. 107–122. DOI: 10.4204/EPTCS.305.8.

[VJ00] J. Vöge and M. Jurdziński. "A Discrete Strategy Improvement Algorithm for Solving Parity Games". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. A. Emerson and A. P. Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 202–215. DOI: 10.1007/10722167_18.

[vR19] T. van Dijk and B. Rubbens. "Simple Fixpoint Iteration To Solve Parity Games". In: *Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2-3rd September 2019*. Ed. by J. Leroux and J.-F. Raskin. Vol. 305. EPTCS. 2019, pp. 123–139. DOI: 10.4204/EPTCS.305.9.

[VW86] M. Y. Vardi and P. Wolper. "An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)". In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 332–344.

[Wal02] I. Walukiewicz. "Monadic second-order logic on tree-like structures". In: *Theor. Comput. Sci.* 275.1-2 (2002), pp. 311–346. DOI: 10.1016/S0304-3975(01)00185-2.

[Wal96] I. Walukiewicz. "Monadic Second Order Logic on Tree-Like Structures". In: *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22-24, 1996, Proceedings*. Ed. by C. Puech and R. Reischuk. Vol. 1046. Lecture Notes in Computer Science. Springer, 1996, pp. 401–413. DOI: 10.1007/3-540-60922-9_33.

[Zie98] W. Zielonka. "Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees". In: *Theor. Comput. Sci.* 200.1-2 (1998), pp. 135–183. DOI: 10.1016/S0304-3975(98)00009-7.

[Zon07] D. Zongker. "Chicken Chicken Chicken". In: *Annals of Improbable Research* 12.6 (2007).

List of publications

[Cra+16] M. Cramer, P. V. Hertum, R. Lapauw, I. Dasseville, and M. Denecker. "Resilient Delegation Revocation with Precedence for Predecessors Is NP-Complete". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. IEEE Computer Society, 2016, pp. 432–442. DOI: 10.1109/CSF.2016.37.

[LBD19] R. Lapauw, M. Bruynooghe, and M. Denecker. *Improving Parity Games with Justifications*. Version 0.3. Oct. 2019. DOI: 10.5281/zenodo.3510292.

[LBD20] R. Lapauw, M. Bruynooghe, and M. Denecker. "Improving Parity Game Solvers with Justifications". In: *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*. Ed. by D. Beyer and D. Zufferey. Vol. 11990. Lecture Notes in Computer Science. Springer, 2020, pp. 449–470. ISBN: 978-3-030-39321-2. DOI: 10.1007/978-3-030-39322-9_21.

[LBDar] R. Lapauw, M. Bruynooghe, and M. Denecker. "Justifications and Reconstructing Parity Game Algorithms". In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems, Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday* (2020 (to appear)).

[LDD15] R. Lapauw, I. Dasseville, and M. Denecker. "Visualising interactive inferences with IDPD3". In: *CoRR* abs/1511.00928 (2015). arXiv: 1511.00928. URL: http://arxiv.org/abs/1511.00928.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Leuven
ruben.lapauw@cs.kuleuven.be
https://www.dtai.cs.kuleuven.be