Synthesis of Distributed Systems

Dissertation zur Erlangung des Grades des Doktors der Naturwissenschaften der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

Sven Schewe

Saarbrücken, 2008

Tag des Kolloquiums	25.07.2008
Dekan	Prof. Dr. Joachim Weickert

Prüfungsausschuss

Vorsitzender	Prof. Dr. Jörg Siekmann
Berichterstattende	Prof. Dr. Bernd Finkbeiner Prof. Dr. Amir Pnueli Prof. Dr. Andreas Podelski Prof. Dr. Moshe Y. Vardi
Akademischer Mitarbeiter	Dr. Andrey Rybalchenko

Abstract

This thesis offers a comprehensive solution of the distributed synthesis problem.

It starts with the problem of solving Parity games, which form an integral part of the automata-theoretic synthesis algorithms we use. We improve the known complexity bound for solving parity games with n positions and c colors approximately from $O(n^{\frac{1}{2}c})$ to $O(n^{\frac{1}{3}c})$, and introduce an accelerated strategy improvement technique that can consider all combinations of local improvements in every update step, selecting the globally optimal combination.

We then demonstrate the decidability and finite model property of alternating-time specification languages, and determine the complexity of the satisfiability and synthesis problem for the alternating-time μ-calculus and the temporal logic ATL*.

The impact of the architecture, that is, the set of system processes with known (white-box) and unknown (black-box) implementation, and the communication structure between them, is determined. We introduce *information forks*, a simple but comprehensive criterion that characterizes all architectures for which the synthesis problem is undecidable. The information fork criterion takes the impact of nondeterminism, the communication topology, and the specification language into account. For decidable architectures, we present an automata-based synthesis algorithm.

We introduce bounded synthesis, which deviates from general synthesis by considering only implementations up to a predefined size, and thus avoids the expensive representation of all solutions. We develop a SAT based approach to bounded synthesis, which is nondeterministic quasilinear in the minimal implementation instead of nonelementary in the system specification.

We determine the complexity of open synthesis under the assumption of probabilistic or reactive environments. Our automata based approach allows for a seamless integration of the new environment models into the uniform synthesis algorithm.

Finally, we study the synthesis problem for asynchronous systems. We show that distributed synthesis remains only decidable for architectures with a single black-box process, and determine the complexity of the synthesis problem for different scheduler types. Furthermore, we combine the undecidability results and synthesis procedures for synchronous and asynchronous systems; systems that are globally asynchronous and locally synchronous are decidable if all black-box components are contained in a single fork-free synchronized component.

Zusammenfassung

Diese Dissertation löst das Syntheseproblem für verteilte Systeme.

Sie beginnt mit verbesserten Algorithmen zum Lösen von Parity Spielen, die einen integralen Bestandteil der Automaten basierten Synthese bilden. Die bekannte Komplexitätsschranke für das Lösen von Parity Spielen mit n Knoten und c Farben wird von ca. $O(n^{\frac{1}{2}c})$ auf ca. $O(n^{\frac{1}{3}c})$ verbessert, und es wird eine beschleunigte Strategie Verbesserungsmethode entwickelt, die, in jedem Schritt, die optimale Kombination aller lokalen Verbesserungen findet.

Die Entscheidbarkeit alternierender Logiken wird gezeigt, und die Komplexität des Erfüllbarkeits- und Syntheseproblems für das Alternierende μ-Kalkül (EXPTIME-vollständig) und die Temporallogik ATL* (2EXPTIME-vollständig) bestimmt.

Der Einfluss der Systemarchitektur, der Spezifikationssprache und, damit verbunden, des Implementierungsmodells (deterministisch vs. nichtdeterministisch) auf die Entscheidbarkeit und Komplexität des Syntheseproblems wird herausgearbeitet. Es wird gezeigt, dass die Klasse der entscheidbaren Architekturen durch die Abwesenheit von *Information Forks*, einem einfachen und leicht prüfbaren Kriterium auf der Kommunikationsarchitektur, vollständig beschrieben werden kann. Für entscheidbare Architekturen wird ein einheitliches Automaten basiertes Syntheseverfahren entwickelt.

Darüber hinaus wird ein SAT basiertes Verfahren entwickelt, dass die Repräsentation aller Lösungen in einem Automaten umgeht. Die Komplexität des SAT basierten Verfahrens ist nichtdeterministisch quasilinear in der Größe des minimalen Modells, statt nicht-elementar in der Größe der Spezifikation.

Für probabilistische und reaktive Umgebungen wird die Komplexität des offenen Syntheseproblems bestimmt, und jeweils ein Automaten basiertes Syntheseverfahren entwickelt, dass sich nahtlos in das Syntheseverfahren für verteilte Systeme integrieren lässt.

Ferner wird gezeigt, dass verteilte Synthese für asynchrone Systeme nur dann entscheidbar bleibt, wenn lediglich die Implementierung einer Komponente konstruiert werden soll. Schließlich werden die Entscheidbarkeitsresultate und Synthese Algorithmen für synchrone und asynchrone Modelle zusammengeführt: Global asynchrone lokal synchrone Systeme sind entscheidbar, wenn alle zu synthetisierenden Prozesse in der gleichen synchronisierten Komponente liegen, und diese Komponente keine Information Forks enthält.

Acknowledgments

There is no way to express my gratitude towards my adviser Bernd Finkbeiner. Bernd has always been the role model of a supervisor. He gave me the most valuable gift an adviser can give to his advisee: A well thought-out thesis topic that lasted for years of intensive and challenging research. The intensity of Bernd's supervision can maybe best described by our ten joint publications, eight of which form the backbone of this thesis; I will draw from our close cooperation for years. Besides this invaluable support, Bernd also gave me trust and independence, and provided me with challenges to grow on. He always believed in me, even when I did not, and I still wonder what made him accept me as a PhD student at a time where I had not even heard of ω-automata or temporal logics. It will be hard to live up to the standards he set – Bernd will always remain the archetype of an adviser that I will try to reach.

It is an honor for me to have Amir Pnueli and Moshe Vardi on my thesis committee. Both are not only leading researchers in the field, they also had a strong influence on this thesis. The paper "Synthesizing Distributed Systems [KV01]" by Orna Kupferman and Moshe Vardi has been the first work I came in contact with when starting my PhD, and their work on automata-theoretic synthesis [Var95, KV97b, Var98, KV99, KV00, KV05] has had a strong influence on my research. When Moshe's work has given me guidance, Amir's work has initialized the area of research I focused on: The paper "Distributed Reactive Systems are Hard to Synthesize [PR90]" of Amir Pnueli and Roni Rosner has been the starting point for distributed synthesis, and prior work of theirs [PR89a, PR89b] has triggered open synthesis for temporal logic.

I am also grateful to Andreas Podelski for many inspiring discussions, and for joining my thesis committee.

I want to thank Rayna Dimitrova, Klaus Dräger, Rüdiger Ehlers, Holger Hermanns, Lars Kuhtz, Anne Proetzsch, Hans-Jörg Peter, Andrey Rybalchenko, Christa Schäfer, Jonathan Türpe, and many others for several fruitful discussions and productive coffee brakes.

I am grateful to the German Research Foundation (DFG) for supporting this work as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), and through the Research School "Quality Guarantees for Computer Systems".

My last thanks go to those who are closest to me. I thank my family – Kasia and Alina – for uncomplainingly allowing much too long working hours, and for the love and support they gave me.

Contents

Chapter 1

Introduction

The influence of computer systems on our life has been constantly growing for several decades, and they seem to gain more and more importance every day. Mal-functional systems often endanger life, both in civil systems – for example, in automotive and air traffic control systems – and in military systems – for example, in command and control systems or missile controllers. Errors may also cause severe financial damage or violation of privacy – for example in transaction systems and security protocols. Consequently, one of the main challenges in computer science is the design of provably correct systems.

Most of these safety critical computer systems and protocols are reactive in nature: Systems of non-terminating processes that interact with each other over an infinite run. Parallelism, nondeterminism, and an incomplete view of the processes on the global system state make it difficult to analyze and design such systems correctly. There are two basic approaches to obtain a provably correct system: Methods which prove that an implementation satisfies a logical property (verification) and algorithms that derive correct-by-construction implementations from logical specifications (synthesis).

While verification has been the subject of intensive research for over 40 years and has become a standard tool in the design of safety critical systems, research on the synthesis problem is still in its infancy.

The topic of this thesis is the automated construction of distributed reactive systems – systems consisting of several independent processes that cooperate based on local information to accomplish a global goal – from their specifications. While verification can only be applied once the design of a system is complete, synthesis algorithms can automatically construct prototypes (which are helpful

to validate the system requirements), analyze partial designs (to determine if it is still possible to complete the implementation to a correct system), and automatically fill in auxiliary components (such as protocol adapters) to systems built from component libraries.

The results of this thesis promote the understanding of the concepts behind the distributed synthesis problem and are a step forward in the process of bringing synthesis to the same level of maturity and industrial acceptance that has currently been reached by verification.

1.1 History of Synthesis

Synthesis algorithms decide whether a given specification has an implementation. For distributed systems, the specification is usually given as an ω-regular set of system behaviors, and an implementation is a collection of finite-state programs that satisfy the formula when composed into a complete system.

The synthesis problem for reactive systems was originally introduced by Church [Chu63] in 1962: Given a relation $R \subseteq (2^I)^\omega \times (2^O)^\omega$ in the monadic second order theory of one successor (S1S), we want to find a function $p : (2^I)^\omega \to (2^O)^\omega$ such that $(\pi, p(\pi)) \in R$ satisfies the relation for all infinite sequences $\pi \in (2^I)^\omega$. In synthesis, we additionally require that the function p is realizable, that is, p shall not depend on the future: The initial sequence of any length $n \in \omega$ of $p(\pi)$ depends only on the initial sequence of the same length of the input sequence π.

Church's problem was solved independently by Rabin [Rab69], and Büchi and Landweber [Büc62, BL69b, BL69a] in 1969. Since their seminal works, the close relation between finite automata over infinite structures [Rab69] and finite games of infinite duration [BL69b, BL69a] became apparent, and both areas have often inspired each other.

More recently, synthesis focused on systems with easy-to-grasp languages like temporal and fixed point logics. The synthesis algorithms in the literature solve various instances of the synthesis problem that differ in the choice of the system architecture and the specification logic. *Closed synthesis*, the case of a single-process implementation without any interaction with the environment, was solved in the early 80's for CTL [CE82], LTL [Wol82] and the modal μ-calculus [Koz83]. More recently, the closed synthesis problem has been solved for the weakest alternating-time logic ATL [vD03, WLWW06]. *Open synthesis* concerns systems consisting of a single process that interacts with its environment, and has been solved for asynchronous systems with

linear specifications [PR89b, Var95] as well as for synchronous systems with branching-time [KV97b] and μ-calculus [KV00] specifications. In *distributed synthesis* [PR90, MT01, KV01], the system is decomposed into different processes that cooperate based on a limited local knowledge about the global system state. An automata-based synthesis algorithm for pipeline and ring architectures and CTL* specifications is due to Kupferman and Vardi [KV01], and Walukiewicz and Mohalik provided an alternative game-based construction [WM03]. There is also a negative result: Pnueli and Rosner [PR90] showed that the synthesis problem is undecidable for LTL specifications and a simple architecture, consisting of two processes that receive incomparable information from an external environment.

1.2 Contribution

This thesis solves the synthesis problem for distributed systems. A distributed system consists of several programs that each implement an independent process. These programs must be constructed in such a way that, even though they each only have partial information about the system state, their joint behavior satisfies the specification. The introduced methods cover all design phases:

- At the beginning of system design, we usually have only a high-level partition of the system into independent processes, but not yet an architecture that reflects the inter-process communication. Alternating-time logics are a natural specification language for this design stage; ATL* or alternating-time μ-calculus (ATμC) specifications reflect the coalition power of the participating processes in a closed system. On this level, we are only interested in what the processes can achieve, but not yet in the question how their cooperation or competition is organized.

  ATL* and ATμC are shown to be decidable, and the complexity of their satisfiability problem and its constructive extension to synthesis are determined.

- In the next design stage, interfaces between the processes are established. The synthesis problem in this design phase is the traditional distributed synthesis problem [PR90, KV01]: The input consists of a temporal specification and an *architecture*. The system architecture is given as a directed graph, whose nodes represent the processes, and whose edges reflect the communication topology (or static interfaces). The edges are labeled with

Boolean variables that serve at the same time for inter-process communication, and as atomic propositions for the specification.

While it is well known that distributed synthesis is undecidable in general [PR90], some architectures, such as pipelines [PR90], chains, and rings [KV01] have been known to be decidable. The borderline between decidable and undecidable architectures, however, remained an open challenge.

This thesis provides a concise description of the class of decidable architectures, and the influence of the actor model (deterministic vs. nondeterministic) and process cooperation (synchronous vs. asynchronous) on the class of decidable architectures.

- In later design stages, we are faced with *partial designs*, where a subset of the system processes has a known implementation. To cover this design phase, the description of the class of decidable architectures is rendered more precisely by distinguishing processes with a known implementation (which are called white-box processes in this thesis) from those with an unknown implementation (black-box processes).

  The effect of turning processes white-box on the decidability of an architecture is analyzed.

Covering these different levels, the suggested methods can be used for finding design errors as soon as they occur. Unrealizable specifications can be distinguished from erroneously constructed interfaces, and errors in the interface design can be distinguished from implementation errors.

Additionally, the introduced techniques can be used for error localization: By replacing a process (or a small set of processes) by black-boxes, we can check if this process can be blamed for the error in the sense that the error can be avoided by changing its implementation.

The core results of the thesis are

- the finite model theorem for alternating-time specification languages and the complexity of their satisfiability problem – the satisfiability and synthesis problem is EXPTIME-complete for the alternating-time μ-calculus and 2EXPTIME-complete for ATL* (Part II); and
- the identification of the fundamental parameters of the distributed synthesis problem and their influence on its decidability and complexity (Part III).

Based on this classification, a general synthesis algorithm that provides a uniform solution for all decidable cases, and for different environment models (maximal, probabilistic, and reactive environments) is suggested in Part III.

Additionally, this thesis covers algorithmic aspects of the synthesis problem. The discussed algorithms are automaton-based, and the most expensive steps in these algorithms are constructive non-emptiness tests of nondeterministic parity automata. Testing their non-emptiness reduces to solving parity games. Performant algorithms for solving parity games are therefore an important step on the way towards applicable synthesis procedures. In Chapter 2, the known complexity bound for solving parity games with c colors and n positions is improved from approximately $O(n^{\frac{1}{2}c})$ to approximately $O(n^{\frac{1}{3}c})$, and a performant strategy improvement method is introduced in Chapter 3.

The complexity of distributed synthesis is nonelementary even for the decidable fragment. In particular, the size of the smallest distributed implementation may be nonelementary in the length of the specification. From a more applied point of view, such solutions are of limited interest, because they cannot be implemented. Inspired by the success of bounded model checking [CFG+01, BCC+03], *bounded synthesis* – the problem of finding a distributed implementation with bounded size – is introduced in Chapter 7. Bounded synthesis has two advantages over traditional synthesis techniques: It can be used to construct *minimal* solutions, and as a semi decision procedure for undecidable architectures.

Finally, a *compositional synthesis rule* is introduced, which establishes the realizability of a specification by showing that the specification can be strengthened into a conjunction of local specifications for the individual processes, such that each local specification is *resiliently* realized by its process (Chapter 9). A process implementation satisfies its specification resiliently if it satisfies its specification no matter how the remainder of the system is implemented. While the specification must be strengthened manually, checking the correctness of the strengthening and constructing resilient implementations can be automated.

1.3 Organization of the thesis

The organization of the thesis follows the technical dependencies. The first part introduces improved algorithms for parity games. The second part refers to the realizability problem for alternating-time logics. Part II also contains several automata transformations that are used in the synthesis procedures of Part III.

In the third part, the synthesis problem for distributed architectures is solved along the four dimensions of the synthesis problem – system architecture, process cooperation, process composition, and environment model.

1.3.1 Part I – Parity Games

The cost of automata-theoretic algorithms for satisfiability checking and synthesis is dominated by the non-emptiness test of nondeterministic parity tree automata, that is, by the cost of solving large parity games. In Part I, two algorithms for solving parity games are discussed, a forward-backward algorithm that combines McNaughton's [McN93] backward technique for solving parity games with an incomplete version of Jurdziński's forward analysis [Jur00]. By balancing the time invested into both parts, the complexity bound for solving parity games with n positions and m edges is improved from $O(m\,n^{\lfloor 0.5\,c \rfloor})$ for the qualitative analysis and $O(m\,n^{\lceil 0.5\,c \rceil})$ for a constructive solution of parity games with c colors [Jur00] to $O(m\,n^{\gamma(c)})$ with $\gamma(c) = \frac{1}{3}c + \frac{1}{2} - \frac{1}{\lfloor 0.5\,c \rfloor \lceil 0.5\,c \rceil}$ for even numbers of colors, and $\gamma(c) = \frac{1}{3}c + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lfloor 0.5\,c \rfloor \lceil 0.5\,c \rceil}$ for odd numbers of colors, respectively. Prior to this result, there have only been two improvements of the complexity bounds since McNaughton introduced his iterated fixed-point approach, and only one of them – the algorithm of Browne, Clarke, Jha, Long, and Marrero [BCJ+97] – restricted the growth of the exponent in the number of colors (from $c-1$ to $\lceil 0.5\,c \rceil + 1$). The following table illustrates the improvement of the complexity for parity games with n positions and m edges obtained by the big step approach introduced in Chapter 2, compared to previous algorithms:

# colors	3	4	5	6	7	8	9
McNaughton	$O(mn^2)$	$O(mn^3)$	$O(mn^4)$	$O(mn^5)$	$O(mn^6)$	$O(mn^7)$	$O(mn^8)$
Browne & al.	$O(mn^3)$	$O(mn^3)$	$O(mn^4)$	$O(mn^4)$	$O(mn^5)$	$O(mn^5)$	$O(mn^6)$
Jurdziński	$O(mn^2)$	$O(mn^2)$	$O(mn^3)$	$O(mn^3)$	$O(mn^4)$	$O(mn^4)$	$O(mn^5)$
Big Steps	$O(mn)$	$O(mn^{1.5})$	$O(mn^2)$	$O(mn^{2.\overline{3}})$	$O(mn^{2.75})$	$O(mn^{3.0625})$	$O(mn^{3.45})$

Additionally, a performant strategy improvement algorithm is introduced that, in every improvement step, considers *all combinations* of strategy improvements. While our experimental data suggests that the proposed strategy improvement algorithm performs extremely well in practice, its complexity is wide open (between quasi trilinear in m, n and c, and $O(m\,n^c)$).

The content of Part I is partially based on the following publications:

[Sch07] Sven Schewe. Solving parity games in big steps. In *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007), 12–14 December, New Delhi, India*, volume 4805 of *Lecture Notes in Computer Science*, pages 449–460. Springer-Verlag, 2007.

[Sch08b] Sven Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proceedings of the 17th Annual Conference of the European Association for Computer Science Logic (CSL 2008), 15–19 September, Bertinoro, Italy*, volume 5213 of *Lecture Notes in Computer Science*, pages 368–383. Springer-Verlag, 2008.

1.3.2 Part II – Logic & Automata

Alternating-time logics extend the scope of distributed systems to systems of independent agents that cooperate on some objectives while competing on others. This thesis studies the alternating-time μ-calculus and the alternating-time temporal logic ATL*, which extend classic μ-calculus and CTL*, respectively, with modalities and path quantifiers that quantify over the strategic choices of a group of agents.

A dedicated type of automata – automata over concurrent game structures – is introduced for the analysis of alternating-time properties. Based on this representation, automata theoretic techniques are exploited to demonstrate the finite model property for ATμC and its semantic sublogic ATL*, and to show that the satisfiability and synthesis problem for ATμC and ATL* are EXPTIME-complete and 2EXPTIME-complete, respectively.

The content of Part II is partially based on the following publications:

[SF06a] Sven Schewe and Bernd Finkbeiner. The alternating-time μ-calculus and automata over concurrent game structures. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic (CSL 2006), 25–29 September, Szeged, Hungary*, volume 4207 of *Lecture Notes in Computer Science*, pages 591–605. Springer-Verlag, 2006.

[Sch08a] Sven Schewe. Sven Schewe. ATL* satisfiability is 2ExpTime-complete. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II (ICALP 2008), 6–13 July, Reykjavik, Iceland*, volume 5126 of *Lecture Notes in Computer Science*, pages 373–385. Springer-Verlag, 2008.

1.3.3 Part III – Distributed Architectures

We study system architectures in a generic setting, where an architecture is given as a directed graph. The nodes represent processes and may include the external environment as a special process. The edges of the graph are labeled with variables, indicating that data may be transmitted between two processes.

The basic concept that distinguishes architectures with a decidable synthesis problem from those, for which the synthesis problem is undecidable, turns out to be the possibility to order the processes that are subject to synthesis with respect to their informedness. This distinction applies to all settings, no matter if the processes are composed synchronously or asynchronously, and independent of the actor model (deterministic vs. nondeterministic processes).

Synchronous Systems. For synchronous systems, we introduce *information forks*, a simple but comprehensive criterion that characterizes all architectures for which the synthesis problem is undecidable. For all fork-free architectures, we provide a uniform synthesis algorithm (Chapter 6). The question if an architecture contains an information fork depends on the set of nondeterministic processes. (Nondeterminism is, for example, necessary to model the leeway needed by different agents to fulfill an alternating-time specification.) For nondeterministic systems, the class of decidable architectures is slightly reduced compared to systems where all processes but an external environment are deterministic: Presuming implicit forwarding of knowledge, which is possible in a deterministic setting, cannot be justified for nondeterministic processes, and the hierarchy of informedness must be directly implied by a subset relation on the information available to the single processes.

A frequent critic to synthesis is that the complexity of synthesis in general, and the seemingly high complexity of distributed synthesis in particular (nonelementary for the decidable fragment, and CoRE in general) makes synthesis infeasible in practice. This high complexity is, however, due to the fact that the smallest implementation may be of nonelementary and unbounded size, respectively, which renders any comparison with model checking – where a particular model is part of the input – unfair. We level the playing field between model checking and synthesis by a twist in the synthesis question: Inspired by the success of bounded model-checking, *bounded synthesis* is introduces in Chapter 7. In bounded synthesis, we are only interested in small implementations, whose size does not exceed predefined bounds. Bounded synthesis comes with various advantages over standard synthesis: It allows us to take additional design constraints like the available memory into account (solutions of nonelementary size

cannot be implemented), and to look for minimal solutions. Also, bounded synthesis is decidable for all architectures, even if they contain an information fork, and – by stepwise increasing the bound – provides a semi-decision procedure for the traditional synthesis question. For fork-free architectures, we can even compute a sufficiently large bound for which traditional and bounded realizability coincides, which turns this approach into a decision procedure.

Historically, the synthesis problem has been studied in the setting of *closed systems*, where the system has no interaction with its environment, and in the setting of *open systems* where the system interacts with a maximal environment, that is, an environment that shows every possible behavior. In this thesis, we investigate two additional settings:

Probabilistic environments (Chapter 8) choose their actions randomly rather than nondeterministically. For such environments we study the problems of satisfying a specification *almost-surely* (that is, with probability 1) and *observably*, that is, with non-zero probability. We show that the problem of synthesizing a process implementation from CTL and LTL specifications is EXPTIME-complete and 2EXPTIME-complete, respectively, for probabilistic environments.

Reactive environments (Chapter 9), introduced by Kupferman and Vardi [KV97a], have the power to disable some of their own actions. Synthesis for reactive environments has so far only been studied in the setting of complete information [KMTV00]. We show that the incomplete information available to an individual process does not affect the complexity of the synthesis problem in reactive environments: For CTL, CTL*, and μ-calculus specifications, single-process synthesis remains 2EXPTIME-complete, 3EXPTIME-complete, and 2EXPTIME-complete, respectively.

In Chapter 9, reactive environments are also used in a compositional synthesis rule for the construction of distributed systems. The proposed compositional synthesis rule requires a manual strengthening of the specification into local specifications, which have to be satisfied resiliently by the individual processes. This step can be viewed as a formalization of the construction of a dynamic interface. It is shown to be complete in the sense that every distributed realizable specification can be strengthened accordingly.

Asynchronous Systems. Most synthesis algorithms assume the processes to run synchronously. The synthesis of asynchronous systems (Chapter 10) is more difficult: While synchronous processes are aware of each change to their input, asynchronous processes may fail to see certain changes (when the writing

process is scheduled more often than the reading process) and may see duplicate input values (when the reading process is scheduled multiple times between two writes).

It turns out that the synthesis problem is decidable for asynchronous distributed systems if and only if at most one process implementation is unknown. The cost of synthesizing a single-process implementation is the same for synchronous and asynchronous systems (2EXPTIME-complete for CTL*, and EXPTIME-complete for CTL and the μ-calculus) if we assume a full scheduler (that is, a scheduler that allows every possible scheduling). Lifting this assumption by requiring that the distributed implementation shall satisfy the specification for all schedulers renders the synthesis problem for asynchronous systems exponentially harder (3EXPTIME-complete for CTL*, and 2EXPTIME-complete for CTL and the μ-calculus).

The decision procedure for asynchronous systems of Chapter 10 can be combined with the decision procedure for synchronous systems of Chapter 6 to a decision procedure for systems that are globally composed asynchronously, but contain islands of synchronized processes (GALS systems [Gup03]). This thesis closes with a proof that GALS systems are decidable if and only if all black-box processes are contained in a single fork-free quotient of synchronized processes.

The content of Part III is partially based on the following publications:

[FS05b] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), 26–29 June, Chicago, Illinois, USA*, pages 321–330. IEEE Computer Society Press, 2005.

[FS05a] Bernd Finkbeiner and Sven Schewe. Semi-automatic distributed synthesis. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005), 4–7 October, Taipei, Taiwan*, volume 3707 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2005.

[Sch06] Sven Schewe. Synthesis for probabilistic environments. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA 2006), 23–26 October, Beijing, China*, volume 4218 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2006.

[SF06b] Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2006), 12–14 July, Venice, Italy*, volume 4407 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2006.

[SF07a] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2007.

[SF07b] Sven Schewe and Bernd Finkbeiner. Distributed synthesis for alternating-time logics. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2007.

[SF07c] Sven Schewe and Bernd Finkbeiner. Semi-automatic distributed synthesis. *International Journal of Foundations of Computer Science*, 18(1):113–138, 2007.

[FS07] Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Proceedings of the 2nd Workshop for Automated Formal Methods (AFM 2007), 6 November, Atlanta, Georgia, USA*, pages 69–76. ACM Press, 2007.

Part I

Parity Games

Overview

The final and most expensive step in the synthesis algorithms proposed in Parts II and III consists of a constructive non-emptiness test for nondeterministic parity tree automata, and thus reduces to solving parity games. The first part of this thesis is therefore dedicated to algorithms for solving parity games. The two chapters in this part have a different focus: While Chapter 2 provides an improved complexity bound for solving parity games, Chapter 3 contains a performant algorithm with unknown complexity.

The algorithm proposed in Chapter 2 joins the strengths of traditional forward and backward techniques. While it follows the structure of McNaughton's iterated fixed point algorithm [McN93], it guarantees big update steps by preceding every recursive call with an alternative paradise construction based on Jurdziński's algorithm [Jur00]. By leveling the cost of this paradise construction and the recursive call we can improve the known complexity bound for solving parity games from $O\big(c\,m\,(\frac{2n}{c})^{\lceil 0.5\,c\rceil}\big)$ to $O\big(m\,\big(\frac{\kappa\,n}{c}\big)^{\gamma(c)}\big)$ for $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lceil \frac{c}{2}\rceil \lfloor \frac{c}{2}\rfloor}$ if c is even, and $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{\lceil \frac{c}{2}\rceil \lfloor \frac{c}{2}\rfloor}$ if c is odd (where κ is a small constant).

Chapter 3 suggests an efficient strategy improvement algorithm that overcomes a critical weakness of prior strategy improvement algorithms: While current strategy improvement algorithms have two distinct phases in which strategy updates are selected and evaluated, respectively, the strategy improvement algorithm proposed in Chapter 3 concurrently evaluates all combinations of (not necessarily strict) local improvements to the strategy, and selects their globally optimal combination. The algorithm has a non-trivial upper bound which is exponential only in the number of colors ($O(m\,n^c)$), but its complexity is wide open: The algorithm is only known to be quasi trilinear in the number of colors, edges, and positions of the game.

Chapter 2

Solving Parity Games in Big Steps

Abstract

In this chapter, a new algorithm with an improved complexity bound for solving parity games is proposed. This algorithm combines McNaughton's iterated fixed point algorithm with a preprocessing step, which is called prior to every recursive call. The preprocessing uses ranking functions similar to Jurdziński's, but with a restricted codomain, to determine all winning regions smaller than a predefined parameter. The combination of the preprocessing step with the recursive call guarantees that McNaughton's algorithm proceeds in big steps, whose size is bounded from below by the chosen parameter. High parameters guarantee small call trees, but to the cost of an expensive preprocessing step. An optimal parameter balances the cost of the recursive call and the preprocessing step, resulting in an $O(m\, n^{\gamma(c)})$ complexity bound for solving parity games with c colors, n positions and m edges, where $\gamma(c)=\frac{c}{3}+\frac{1}{2}-\frac{1}{3c}-\frac{1}{\lceil \frac{c}{2} \rceil \lfloor \frac{c}{2} \rfloor}$ if c is even, and $\gamma(c)=\frac{c}{3}+\frac{1}{2}-\frac{1}{\lceil \frac{c}{2} \rceil \lfloor \frac{c}{2} \rfloor}$ if c is odd.

2.1 Introduction

Parity games have many applications in model checking [Koz83, EJS93, KV97a, dAHM01, Wil01, AHK02] and synthesis [Koz83, Var98, Wil01, Pit06]. In par-

ticular, modal and alternating-time μ-calculus model checking [Wil01, AHK02], synthesis [WM03, Pit06] and satisfiability checking [Wil01, Koz83, Var98] for reactive systems, module checking [KV97a], and ATL* model checking [dAHM01, AHK02] can be reduced to solving parity games. In the context of this thesis, solving the non-emptiness test for deterministic and nondeterministic parity tree automata (Chapter 4) – and its applications in the synthesis and satisfiability checking algorithms of Parts II and III – reduce to solving parity games. This relevance of parity games led to a serial of different approaches to solving them [McN93, EL86, Lud95, Pur95, ZP96, BCJ+97, Zie98, Jur98, Jur00, VJ00, Obd03, Lan05, BDHK06, JPZ06, BV07].

The complexity of solving parity games is still an open problem. The known complexity bounds of current deterministic or randomized algorithms are at least exponential in the number of colors [McN93, EL86, ZP96, BCJ+97, Zie98, Jur00, BV07] ($n^{O(c)}$), or in the squareroot of the number of game positions [Lud95, JPZ06, BV07] ($n^{O(\sqrt{n})}$). Practical considerations suggest to assume that the number of colors is small compared to the number of positions. Indeed, all listed applications but μ-calculus model checking result in parity games where the number of states is exponential in the number of colors. In μ-calculus model checking, the size of the game is determined by the product of the transition system under consideration (which is usually large), and the size of the formula (which is usually small). The number of colors is determined by the alternation depth of the specification, which, in turn, is usually small compared to the specification itself.

Algorithms that are exponential only in the number of colors are thus considered the most attractive. The first representatives of algorithms in this complexity class follow the iterated fixed point structure induced by the parity condition [McN93, EL86, Zie98]. The iterated fixed point construction leads to a time complexity of $O(m\,n^{c-1})$ for parity games with m edges, c colors, and n game positions. This complexity could be reduced first by Browne et al. [BCJ+97] to $O\big(mn^{\lceil 0.5c\rceil+1}\big)$, and slightly further by Jurdziński [Jur00] to $O\big(c\,m\,(\frac{2n}{c})^{\lfloor 0.5\,c\rfloor}\big)$ for constructing the winning regions, and $O\big(c\,m\,(\frac{2n}{c})^{\lceil 0.5\,c\rceil}\big)$ for the constructive extension that also provides winning strategies for both players.

The weakness of recursive algorithms that follow the iterated fixed point structure [McN93, EL86, Zie98] is the potentially incremental update achieved by each recursive call. Recently, a big-step approach [JPZ06] has been proposed to reduce the complexity of McNaughton's algorithm for games with a high number of colors ($c \in \omega(\sqrt{n})$) to the bound $n^{O(\sqrt{n})}$ known from randomized algorithms [Lud95, BV07].

In this chapter a novel big-step approach is introduced that improves the complexity for the relevant lower end of the spectrum of colors, resulting in the complexity $O\big(m\,\big(\frac{\kappa\,n}{c}\big)^{\gamma(c)}\big)$ for solving parity games, where κ is a small constant and $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lceil \frac{c}{2} \rceil \lfloor \frac{c}{2} \rfloor}$ if c is even, and $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{\lceil \frac{c}{2} \rceil \lfloor \frac{c}{2} \rfloor}$ if c is odd. At the same time, the deterministic $n^{O(\sqrt{n})}$ complexity bound is preserved.

To guarantee big update steps, we use an algorithm which is inspired by Jurdziński's [Jur00] approach for solving parity games. His approach is adapted by restricting the codomain of the used ranking function. The resulting algorithm is exploited for finding winning regions up to the size of a parameter π. Compared to [JPZ06], this results in a significant cut in the complexity for finding all paradises (winning regions, for which a player can win while staying in the paradise), since the running time of our algorithm is polynomial in the parameter, and exponential only in the number of colors ($O\big(\binom{\pi + \lceil 0.5c \rceil}{\pi}\big)$). As a side effect, the proposed approximation algorithm often returns winning regions of greater size. The claimed complexity bound is obtained by using a parameter of approximately $\sqrt[3]{c\,n^2}$.

2.2 Parity Games

A parity game is composed of a finite arena and an evaluation function. We will first discuss arenas, and then turn to the evaluation functions for parity games.

Arena. Parity games are played on finite arenas. An *arena* is a triple $\mathcal{A} = (V_0, V_1, E)$, where V_0 and V_1 are disjoint finite sets of positions, called the positions of player 0 and 1, respectively, and $E \subseteq V \times V$ for the set $V = V_0 \uplus V_1$ of game positions is a set of edges; that is, (V, E) is a directed graph. The arena is also required not to contain sinks; that is, every position $p \in V$ has at least one outgoing edge $(p, p') \in E$.

Plays. Intuitively, a game is played by placing a pebble on the arena. If the pebble is on a position $p \in V_0$, player 0 chooses an edge $e = (p, p') \in E$ from p to a successor p' and moves the pebble to p'. Symmetrically, if the pebble is on a position $q \in V_1$, player 1 chooses an edge $e' = (q, q') \in E$ from q to a successor q' and moves the pebble to q'. This way, they successively construct an infinite *play* $\pi = p_0 p_1 p_2 p_3 \ldots \in V^\omega$.

Strategies. For a finite arena $\mathcal{A} = (V_0, V_1, E)$, a *strategy* for player 0 is a function $f : V^*V_0 \to V$ which maps each finite history of a play that ends in a position $p \in V_0$ to a successor p' of p. (That is, there is an edge $(p, p') \in E$ from p to p'.) A play is *f-conform* if every decision of player 0 in the play is in accordance with f. A strategy is called *memoryless* if it only depends on the current position. A memoryless strategy for player 0 can be viewed as a function $f : V_0 \to V$ such that $(p, f(p)) \in E$ for all $p \in V_0$. For a memoryless strategy f, we denote with $\mathcal{A}_f = (V_0, V_1, E_f)$ the arena obtained from $\mathcal{A}$ by deleting the transitions from positions of player 0 that are not in accordance with f. ($\mathcal{A}_f$ defines a directed graph where all positions of player 0 have outdegree 1.) The analogous definitions are made for player 1.

Parity Games. A *parity game* is a game $\mathcal{P} = (V_0, V_1, E, \alpha)$ with arena $\mathcal{A} = (V_0, V_1, E)$ and a surjective coloring function $\alpha : V \to \mathcal{C} \subset \mathbb{N}$ that maps each position of $\mathcal{P}$ to a natural number. $\mathcal{C}$ denotes the finite set of colors. For technical reasons we assume that the minimal color of a parity game is $0 = min\{\mathcal{C}\}$, and that $\mathcal{C}$ is an initial sequence of the integers.

Each play is evaluated by the highest color that occurs infinitely often. Player 0 wins a play $\pi = p_0p_1p_2p_3\ldots$ if the highest color occurring infinitely often in the sequence $\alpha(\pi) = \alpha(p_0)\alpha(p_1)\alpha(p_2)\alpha(p_3)\ldots$ is even, while player 1 wins if the highest color occurring infinitely often in $\alpha(\pi)$ is odd.

A strategy f of player 0 (player 1) is called *p-winning* if all f-conform plays starting in p are winning for player 0 (player 1). A position p in $\mathcal{P}$ is p-winning for player 0 (player 1) if player 0 (player 1) has a p-winning strategy. We call the p-winning positions for player 0 (player 1) the *winning region* of player 0 (player 1). Parity games are memoryless determined:

Theorem 2.1 *[McN93] For every parity game $\mathcal{P}$, the game positions are partitioned into a winning region W_0 of player 0 and a winning region W_1 of player 1. Moreover, player 0 and 1 have memoryless strategies that are p-winning for every position p in their respective winning region.* □

In the remainder of this chapter, all strategies are memoryless.

The common intersection and subtraction operations on digraphs are extended to parity games. ($\mathcal{P} \cap F$ and $\mathcal{P} \setminus F$ thus denote the parity games resulting by restricting the arena $\mathcal{A}$ of $\mathcal{P}$ to $\mathcal{A} \cap F$ and $\mathcal{A} \setminus F$, respectively.)

Remark. The restriction that the minimal color is 0 is only technical. If no position with color 0 exists, then we can reduce all colors by 1 and change the

Procedure *McNaughton*($\mathcal{P}$):

1. set c to the highest color occurring in $\mathcal{P}$
2. if $c = 0$ or $V = \emptyset$ then return $(V, \emptyset)$
3. set σ to $c \mod 2$
4. set $W_{1-\sigma}$ to $\emptyset$
5. repeat
   (a) set $\mathcal{P}'$ to $\mathcal{P} \setminus \sigma$-*Attractor*$(\alpha^{-1}(c), \mathcal{P})$
   (b) set (W'_0, W'_1) to *McNaughton*($\mathcal{P}'$)
   (c) if $(W'_{1-\sigma} = \emptyset$ then
      i. set W_σ to $V \setminus W_{1-\sigma}$
      ii. return (W_0, W_1)
   (d) set $W_{1-\sigma}$ to $W_{1-\sigma} \cup (1-\sigma)$-*Attractor*$(W'_{1-\sigma}, \mathcal{P})$
   (e) set $\mathcal{P}$ to $\mathcal{P} \setminus (1-\sigma)$-*Attractor*$(W'_{1-\sigma}, \mathcal{P})$

Figure 2.1: The algorithm *McNaughton*($\mathcal{P}$) returns the ordered pair (W_0, W_1) of winning regions of the players 0 and 1, respectively. V and α denote the states and the coloring function of the parity game $\mathcal{P}$.

roles of player 0 and 1. Winning regions and strategies for player 0 (player 1) in the resulting game are the winning regions and strategies for player 1 (player 0) in the original game.

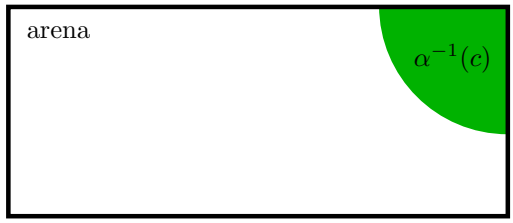
2.3 Established Algorithms

This section shortly recapitulates the two established algorithms of McNaughton [McN93, EL86, Zie98] and Jurdziński [Jur00] for solving parity games, from which the algorithm discussed in Section 2.4 draws.

2.3.1 McNaughton's Algorithm

As a base case, parity games that contain only positions with color 0 are easy to solve: Player 0 wins with every strategy from every position.

For general parity games $\mathcal{P}$ with highest color c, McNaughton's algorithm (Figure 2.1) first determines the set $\alpha^{-1}(c)$ of positions with maximal color.

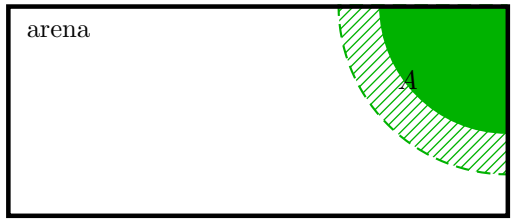


For the player $\sigma = c \mod 2$ that wins if c occurs infinitely often, his algorithm then constructs the σ-attractor A of $\alpha^{-1}(c)$. The *σ-attractor* of a set F of game positions is, for $\sigma \in \{0,1\}$, the set of those game positions, from which player σ has a memoryless strategy to force the pebble to a position in F. The σ-attractor A of a set F can be defined as the least fixed point of sets that contain F, and that contain a game position p of player σ $(1-\sigma)$ in A if it contains some successor (all successors) of p.

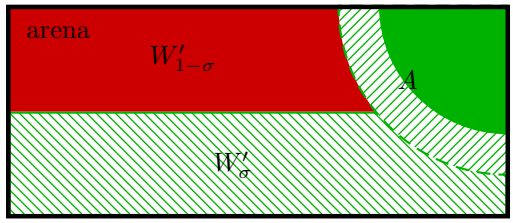
$$\sigma\text{-}Attractor(F,\mathcal{P}) = \bigcap\{S \supset F \mid \forall p \in V_\sigma \forall p' \in S.\,(p,p') \in E \Rightarrow p \in S \text{ and } \forall p \in V_{1-\sigma}.\,(\neg\exists p' \notin S.\,(p,p') \in E) \Rightarrow p \in S\}.$$

Constructing this least fixed point is obviously linear in the number of edges of the parity game, and we can fix a memoryless strategy (the attractor strategy) for player σ to reach F in finitely many steps during this construction.

Lemma 2.2 *For a given parity game $\mathcal{P} = (V_0, V_1, E, \alpha)$, and a set F of game positions, we can compute the σ-attractor A of F and a memoryless strategy for σ on $A \setminus F$ to reach F in finitely many steps in time $O(m)$.* □



In the next step, the co-game $\mathcal{P}' = \mathcal{P} \setminus A$ of $\mathcal{P}$ is solved. The co-game $\mathcal{P}'$ is simpler than $\mathcal{P}$: Compared to $\mathcal{P}$, it contains less positions, and less colors. By induction over the size of the game, $\mathcal{P}'$ can therefore be solved by a recursive call of the algorithm.



We call a subset $P_\sigma \subseteq W_\sigma$ of a winning region a *σ-paradise* if player $\sigma \in \{0,1\}$ has a memoryless strategy f that is p-winning for all $p \in P_\sigma$, such that P_σ cannot be left in any f-conform play ($E_f \cap P_\sigma \times V \setminus P_\sigma = \emptyset$).

In particular, the winning region $W'_{1-\sigma}$ of $\mathcal{P}'$ is a $(1-\sigma)$-paradise in $\mathcal{P}$ by construction – every winning strategy for player $1-\sigma$ on her winning region in $\mathcal{P}'$ is a winning strategy for $W'_{1-\sigma}$ in $\mathcal{P}$ that is guaranteed to stay in $W'_{1-\sigma}$.

For a given σ-paradise P_σ for player $\sigma \in \{0,1\}$ in a parity game $\mathcal{P}$, we can reduce solving $\mathcal{P}$ to computing the σ-attractor A of P_σ, and solving $\mathcal{P} \setminus A$.

Lemma 2.3 *Let $\mathcal{P}$ be a parity game, P_σ a σ-paradise with σ-attractor A, f_σ a winning strategy of σ on P_σ in $\mathcal{P} \cap P_\sigma$, f'_σ the attractor strategy of σ, and let f''_σ and $f_{1-\sigma}$ be winning strategies for σ and her opponent $1-\sigma$, respectively, for their respective winning region in $\mathcal{P} \setminus A$. Then g_σ with $g_\sigma(p) = f_\sigma(p)$, $f'_\sigma(p)$, and $f''_\sigma(p)$ for game positions of player σ in P_σ, $A \setminus P_\sigma$, and not in A, respectively, and $g_{1-\sigma}$ with $g_{1-\sigma}(p) = f_{1-\sigma}(p)$ for all game positions of player $1-\sigma$ not in A, are winning strategies for σ and $1-\sigma$ on their complete winning regions in $\mathcal{P}$.*

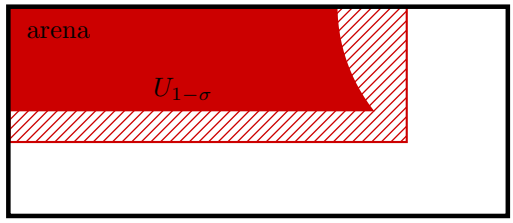
Proof: First player $1-\sigma$ wins with her strategy on her complete winning region $W_{1-\sigma}$ of $\mathcal{P} \setminus A$, since player σ has no additional choices in $W_{1-\sigma}$ in $\mathcal{P}$. That is, the set of $g_{1-\sigma}$-conform plays in $\mathcal{P}$ starting in $W_{1-\sigma}$ coincides with the set of $f_{1-\sigma}$-conform plays in $\mathcal{P} \setminus A$ starting in $W_{1-\sigma}$.

For the same reason, player σ wins with his strategy from every position in P_σ.

Every g_σ-conform play in $\mathcal{P}$ starting in a position in $A \setminus P_\sigma$ consists of a finite prefix in $A \setminus P_\sigma$, followed by a g_σ-conform play in $\mathcal{P}$ starting in P_σ, which we showed to be winning for player σ.

Every g_σ-conform play in $\mathcal{P}$ starting in a position not in A either eventually reaches A, and then is followed by a g_σ-conform play in $\mathcal{P}$ starting in A, which we showed to be winning for σ, or stays for ever in the subgame $\mathcal{P} \setminus A$. If the game started on a position which is winning for σ in $\mathcal{P} \setminus A$, a game that stays in $\mathcal{P} \setminus A$ coincides with an f''_σ-conform play in $\mathcal{P} \setminus A$, which is winning for player σ. □

We now distinguish two cases: Firstly, if $W'_{1-\sigma}$ is nonempty, we can reduce solving $\mathcal{P}$ to constructing the $(1-\sigma)$-attractor $U_{1-\sigma}$ of $W'_{1-\sigma}$, and solving the co-game $\mathcal{P}'' = \mathcal{P} \setminus U_{1-\sigma}$ by Lemma 2.3.



The co-game $\mathcal{P}''$ is simpler than $\mathcal{P}$: Compared to $\mathcal{P}$, it contains less positions (but not necessarily less colors). By induction over the size of the game, $\mathcal{P}''$ can therefore be solved by a recursive call of the algorithm.

Secondly, if $W'_{1-\sigma}$ is empty, we can compose the winning strategy for player σ on $\mathcal{P}'$ with his attractor strategy for $\alpha^{-1}(c)$ to a winning strategy on $\mathcal{P}$.

Lemma 2.4 *Let $\mathcal{P}$ be a parity game with maximal color c, $\sigma = c \mod 2$ the player that wins if c occurs infinitely often, let A be the σ-attractor of $\alpha^{-1}(c)$ and let f be an attractor strategy for player σ on her positions on $A \setminus \alpha^{-1}(c)$. If player σ has a winning strategy f' for every position in $\mathcal{P}' = \mathcal{P} \setminus A$ then f and f' can be composed to a winning strategy for player σ for every position in $\mathcal{P}$.*

Proof: Let g be a strategy for player σ that agrees with f and f' on their respective codomain. We distinguish two types of g-conform plays: Those that eventually stay in $\mathcal{P}'$, and those that visit A infinitely often. The latter plays contain infinitely many c-colored positions and are therefore winning for player σ. Games that eventually stay in $\mathcal{P}'$ consist of a finite prefix, followed by an f'-conform play in $\mathcal{P}'$. The highest color occurring infinitely often is therefore even for $\sigma = 0$ and odd for $\sigma = 1$, respectively. □

The worst case running time of McNaughton's algorithm [McN93, EL86, Zie98] (cf. Procedure *McNaughton* of Figure 2.1) occurs if $U_{1-\sigma}$ is always small and contains exactly one position with maximal color c. For parity games with d colors, McNaughton's algorithm requires $O(m\, n^{d-1})$ steps for games with n states and m edges. It can be extended to also return the winning strategies for both players on their complete winning region.

2.3.2 Progress Measures

The approximation technique introduced in Subsection 2.4.2 builds on Jurdziński's algorithm [Jur00] for solving parity game. His techniques are adapted by restricting the codomain of the ranking function. Some of the theorems stated in this subsection are thus slightly more general than the theorems in [Jur00], but they are arranged such that the proofs provided in [Jur00] can be applied without changes.

For a parity game $\mathcal{P} = (V_0, V_1, E, \alpha)$ with maximal color d, a σ *progress measure* is, for $\sigma \in \{0, 1\}$, a function $\rho : V_0 \uplus V_1 \to \mathcal{M}^\sigma$ whose codomain

$$\mathcal{M}^\sigma \subseteq \{f : \{0, \ldots, d\} \to \mathbb{N} \mid f(c) = 0 \text{ if } c \mod 2 = \sigma, \text{ and } f(c) \leq |\alpha^{-1}(c)| \text{ otherwise}\} \cup \{\top\}$$

contains a maximal element $\top$ and a set of functions from $\{0, \ldots, d\}$ to the integers. The codomain $\mathcal{M}^\sigma$ satisfies the requirement that every integer $i \leq d$ is mapped to 0 if $i \mod 2 = \sigma$, while all other integers i are mapped to a value bounded by the number $|\alpha^{-1}(i)|$ of i-colored game positions. (Jurdziński uses the maximal codomain $\mathcal{M}^\sigma_\infty$ defined by replacing containment with equality.) For simplicity, we require downward closedness: If $\mathcal{M}^\sigma$ contains a function $f \in \mathcal{M}^\sigma$, then every function f' that is pointwise smaller than f ($f'(i) \leq f(i)\ \forall i \leq d$) is also contained in $\mathcal{M}^\sigma$.

For each color $c \leq d$, we define a relation $\rhd_c \subseteq \mathcal{M}^\sigma \times \mathcal{M}^\sigma$. $\rhd_c$ is the smallest relation that contains $\{\top\} \times \mathcal{M}^\sigma$ and a pair of functions $(f, f') \in \rhd_c$ if

- there is a color $c' \geq c$ such that $f(c') > f'(c')$, and $f(c'') = f'(c'')$ holds true for all colors $c'' > c'$, or
- $c \mod 2 = \sigma$, and $f(c') = f'(c')$ holds true for all $c' \geq c$.

That is, $\rhd_c$ is defined by using the lexicographic order, ignoring all colors smaller than c. f needs to be greater than f' by this order, and strictly greater if $c \mod 2 \neq \sigma$.

$\rhd_0$ defines an order $\preceq$ on $\mathcal{M}^\sigma$ (the lexicographic order). From this order, we infer the linear preorder $\sqsubseteq$ on progress measures, which requires that $\preceq$ is satisfied pointwise ($\rho \sqsubseteq \rho' \Leftrightarrow \forall p \in V.\, \rho(p) \preceq \rho'(p)$).

We call a σ-progress measure ρ *valid* if

- every position $p \in V_\sigma$ has some successor $p' \in V$ with $\rho(p) \rhd_{\alpha(p)} \rho(p')$, and
- for every position $p \in V_{1-\sigma}$ and every successors $p' \in V$ of p, $\rho(p) \rhd_{\alpha(p)} \rho(p')$ holds true.

Progress measures are ranking functions, which are intuitively used to estimate the worst-case future occurrence of 'bad' states prior to states with higher color. A valid σ progress measure that is not constantly $\top$ can be used to partly evaluate a parity game. Let, for a σ progress measure ρ, $\|\rho\| = V \setminus \rho^{-1}(\top)$ denote the game positions that are not mapped to the maximal element $\top$ of $\mathcal{M}^\sigma$.

Theorem 2.5 *[Jur00] Let $\mathcal{P} = (V_0, V_1, E, \alpha)$ be a parity game with valid σ progress measure ρ. Then player σ wins on $\|\rho\|$ with any memoryless winning strategy that maps a position $p \in \|\rho\| \cap V_\sigma$ to a position p' with $\rho(p) \rhd_{\alpha(p)} \rho(p')$.*

Such a successor must exist, since the progress measure is valid. The $\sqsubseteq$-*least* valid σ progress measure is well defined and can be computed efficiently.

Theorem 2.6 *[Jur00] The $\sqsubseteq$-least valid σ progress measure ρ_μ exists and can, for a parity game with m edges and c colors, be computed in time $O(c\,m\,|\mathcal{M}^\sigma|)$*[1].

When using the maximal codomain $\mathcal{M}^\sigma_\infty$, which contains the function ρ that assigns each color c with $c \mod 2 \neq \sigma$ to $\rho(c) = |\alpha^{-1}(c)|$, for the progress measures, the $\sqsubseteq$-least valid σ progress measure ρ_μ determines the complete winning region of player σ.

Theorem 2.7 *[Jur00] For a parity game $\mathcal{P} = (V_0, V_1, E, \alpha)$, and for the codomain $\mathcal{M}^\sigma_\infty$ for the progress measures, $\|\rho_\mu\|$ coincides with the winning region W_σ of player σ for the $\sqsubseteq$-least valid σ progress measure ρ_μ.*

For parity games with c colors, the size $|\mathcal{M}^\sigma_\infty|$ of the maximal codomain can be estimated by $(\frac{n}{\lfloor 0.5c \rfloor})^{\lfloor 0.5c \rfloor} + 1$ if c is even, and by $(\frac{n}{\lceil 0.5c \rceil})^{\lceil 0.5c \rceil} + 1$ if c is odd.

Corollary 2.8 *[Jur00] Parity games with maximal color 2 can be solved and a winning winning strategy for player 0 can be constructed in time $O(m\,n)$.*

While a partition into the respective winning regions of both players and a winning strategy for σ on her winning region can easily be inferred from the $\sqsubseteq$-least valid σ progress measure ρ_μ, we cannot infer the winning strategy of her opponent on his winning region from this least fixed point.

[1] More precisely, the complexity is $O(c\,m\,|\mathcal{M}^\sigma|\,\log m)$, where the logarithmic factor stems from the comparison of integers $\leq m$. However, this logarithmic factor does not change the complexity, because address calculation causes the same computational overhead. For this reason, all complexity estimations in this part are sloppy with respect to this logarithmic factor.

Procedure *Winning-Regions*($\mathcal{P}$):

1. set d to the highest color occurring in $\mathcal{P}$
2. if $d \leq 2$ then return *ThreeColor*($\mathcal{P}$)
3. set σ to $d \mod 2$
4. set n to the size $|V|$ of $\mathcal{P}$
5. set $W_{1-\sigma}$ to $\emptyset$
6. repeat
   (a) set $W'_{1-\sigma}$ to $(1-\sigma)$-*Attractor*(*Approximate*($\mathcal{P}, \pi(n,d), 1-\sigma), \mathcal{P}$)
   (b) set $W_{1-\sigma}$ to $W_{1-\sigma} \cup W'_{1-\sigma}$
   (c) set $\mathcal{P}$ to $\mathcal{P} \setminus W'_{1-\sigma}$
   (d) set $\mathcal{P}'$ to $\mathcal{P} \setminus \sigma$-*Attractor*($\alpha^{-1}(d), \mathcal{P}$)
   (e) set (W'_0, W'_1) to *Winning-Regions*($\mathcal{P}'$)
   (f) if $W'_{1-\sigma} = \emptyset$ then
      i. set W_σ to $V \setminus W_{1-\sigma}$
      ii. return (W_0, W_1)
   (g) set $W_{1-\sigma}$ to $W_{1-\sigma} \cup (1-\sigma)$-*Attractor*($W'_{1-\sigma}, \mathcal{P}$)
   (h) set $\mathcal{P}$ to $\mathcal{P} \setminus (1-\sigma)$-*Attractor*($W'_{1-\sigma}, \mathcal{P}$)

Figure 2.2: The Procedure *Winning-Regions*($\mathcal{P}$) returns the ordered pair (W_0, W_1) of winning regions for player 0 and player 1, respectively. V and α denote the game positions and the coloring function of the parity game $\mathcal{P}$. *ThreeColor*($\mathcal{P}$) solves a three color game $\mathcal{P}$ (c.f. Theorem 2.9), *Approximate*($\mathcal{P}, \pi, \sigma$) computes a $\sigma/(\pi+1)$-paradise (c.f. Corollary 2.11), and σ-*Attractor*($F, \mathcal{P}$) computes the σ-attractor of a set F of game positions in a game $\mathcal{P}$ (c.f. Lemma 2.2).

2.4 The Algorithm

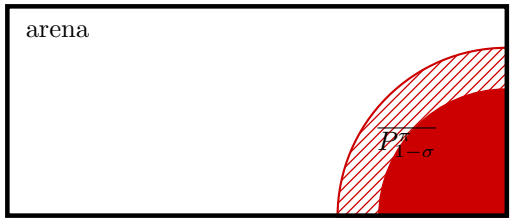
The algorithm proposed in this chapter combines McNaughton's iterated fixed point approach for solving parity games [McN93, EL86, Zie98] with an alternative paradise construction of a special kind of paradises. We call a σ-paradise P^π_σ a *σ/π-paradise* if it contains all σ-paradises of size $\leq \pi$.

Figure 2.2 provides an overview on the proposed algorithm. The input to the algorithm is a parity game $\mathcal{P}$, and the output is the ordered pair consisting of the winning regions for the players.

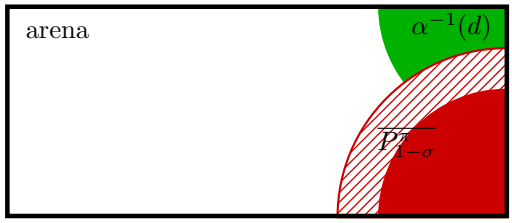
The algorithm first determines the highest color d of $\mathcal{P}$ (line 1). In line 2, three color games are covered, that is, games with highest color ≤ 2. Such games are solved using a constructive extension of Jurdziński's algorithm (c.f. Theorem 2.9 of Subsection 2.4.1). For games with more than 2 colors, the algorithm proceeds with determining the player $\sigma = d \mod 2$ that wins if the highest color d occurs infinitely often (line 3).

In every iteration of the repeat loop, the proposed big step algorithm (Figure 2.2) first constructs a $(1-\sigma)/\pi$-paradise (cf. Subsection 2.4.2) for an appropriate parameter π.

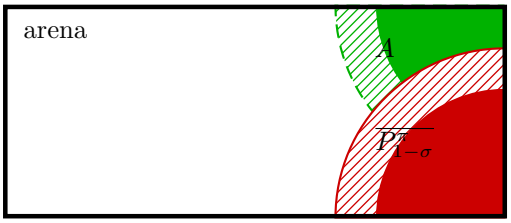
By Lemma 2.3, we can now reduce solving $\mathcal{P}$ to constructing the $(1-\sigma)$-attractor $\overline{P^{\pi}_{1-\sigma}}$ of $P^{\pi}_{1-\sigma}$ (line 6a), and to solving $\mathcal{P}' = \mathcal{P} \setminus \overline{P^{\pi}_{1-\sigma}}$.



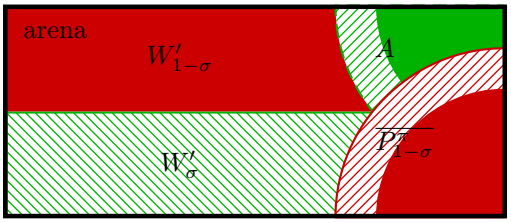
The algorithm then continues with the steps known from McNaughton's algorithm. That is, it next determines the set $\alpha^{-1}(d)$ of positions with maximal color in $\mathcal{P}'$,



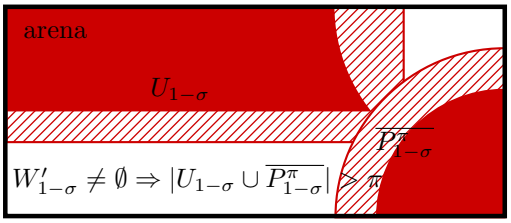
and then constructs the σ-attractor A of $\alpha^{-1}(d)$ in $\mathcal{P}'$ (line 6d).



In the next step, the co-game $\mathcal{P}'' = \mathcal{P}' \setminus A$ of $\mathcal{P}'$ is solved by a recursive call of Procedure *Winning-Regions* (line 6*e*).



If $W'_{1-\sigma}$ is empty, we can again evaluate the game immediately by Lemma 2.4 (line 7*f*). If $W'_{1-\sigma}$ is nonempty, we can reduce solving $\mathcal{P}'$ to constructing the $(1-\sigma)$-attractor $U_{1-\sigma}$ of $W'_{1-\sigma}$, and to solving $\mathcal{P}'' = \mathcal{P}' \setminus W'_{1-\sigma}$ (line 6*h*) by Lemma 2.3.



While we know little about the size of $\overline{P^{\pi}_{1-\sigma}}$ (which may be empty) and $W'_{1-\sigma}$ (which may be singleton), we know that their union is greater than π, because their union is a $(1-\sigma)$-paradise (as the union of two $(1-\sigma)$-paradises), and would otherwise be contained in $P^{\pi}_{1-\sigma}$.

We can therefore impose an upper bound on the number of iterations, which depends on the size of the parameter. While bigger parameters slow down the approximation procedure (c.f. Corollary 2.11), they restrict the size of the call tree.

For reasonable numbers of colors (that is, if the number of colors is in $O(\sqrt{n})$), the best results are obtained if the parameter is chosen such that

the cost of calling the approximation procedure (line 6a) and the cost of the recursive call (line 6e) are approximately equivalent. This is the case if we set the parameter approximately to $\sqrt[3]{c\,n^2}$. (The function β defined for the proof of the complexity quickly converges to $\frac{2}{3}$.)

For a high number of colors (that is, if the number of colors is in $\omega(\sqrt{n})$), the best results are obtained if the cost of calling the approximation procedure (line 6a) approximately coincides with the size of the call tree.

In the remainder of this section, we discuss

1. a constructive algorithm for solving three color games,
2. an algorithm for the efficient construction of σ/π-paradises,
3. a correctness proof for the overall algorithm, and
4. a complexity analysis for suitable parameters for a reasonable and high number of colors, respectively.

2.4.1 Three Color Games

We call parity games with maximal color 2 *three color games*. Corollary 2.8 shows that a qualitative solution for three color games as well as a winning strategy for player 0 can be obtained in time $O(m\,n)$. To see why Jurdziński's algorithm [Jur00] does not provide a strategy for player 1, let us recapitulate his algorithm for the simple case of a three color games $\mathcal{P} = (V_0, V_1, E, \alpha)$.

For three color games, the 0 progress measures can be viewed as mappings $\rho : V \to \{0, \ldots, n_1\} \cup \{\top\}$, where $n_1 = |\alpha^{-1}(1)|$ denotes the number of 1-colored positions. For a given progress measure ρ, we call an edge (p, p') a *lift-edge* if $\rho(p) \not\trianglerighteq_{\alpha(p)} \rho(p')$. We call a position $p \in V_0$ of player 0 *liftable* if all outgoing edges are lift edges, and we call a position $p \in V_1$ of player 1 liftable if some outgoing edge is a lift edge.

Starting point of the algorithm is the trivial progress measure ρ_0 that maps all positions of $\mathcal{P}$ to 0. Starting from ρ_0, we *lift* the progress measure stepwise at a liftable position $p \in V$ until a fixed point is reached. For liftable positions $p \in V_0$ of player 0, lifting ρ_i at p results in a progress measure ρ_{i+1} with $\rho_i(p') = \rho_{i+1}(p')$ for all $p' \neq p$, and

$$\rho_{i+1}(p) = min\{j \in \{0, \ldots, n_1\} \cup \{\top\} \mid \forall (p, p') \in E.\, j \trianglerighteq_{\alpha(p)} \rho(p')\},$$

while for liftable positions $p \in V_1$ of player 1, lifting ρ_i at p results in a progress measure ρ_{i+1} with $\rho_i(p') = \rho_{i+1}(p')$ for all $p' \neq p$, and

$$\rho_{i+1}(p) = min\{j \in \{0, \ldots, n_1\} \cup \{\top\} \mid \exists (p, p') \in E.\, j \trianglerighteq_{\alpha(p)} \rho(p')\}.$$

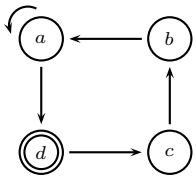


Figure 2.3: The example shows a singleton Büchi game, where all positions belong to player 1 ($V_0 = \emptyset$). The positions a, b, and c are colored by 1, while position d is colored by 2 (indicated by the double line).

For the trivial progress measure ρ_0, an edge is a lift-edge if and only if it originates from a 1-colored position, and a position is liftable if and only if it is 1-colored. For an efficient implementation, it suffices to attach a flag to every edge that indicates whether this edge is a lift-edge, to keep track of the number of outgoing lift-edges for every game position, and to keep the liftable positions in a doubly linked list.

In order to lift ρ_i, any liftable position p can be taken from the list of liftable positions. (If no liftable position remains, the least fixed point is reached.) After lifting ρ_i, at position p, it suffices to check for each incoming and outgoing edge of p if the flag that indicates liftability needs to be adjusted, and, if so, to increase the number of outgoing lift-edges for the respective predecessor of p (for incoming edges), or to decrease the number of outgoing lift-edges for p (for outgoing edges), respectively. If a position becomes liftable (non-liftable), it is added to (removed from) the list of liftable positions.

While this algorithm provides good complexity bounds for the qualitative analysis of three color games, it does not provide a winning strategy for player 1 on her winning region. Note that the naive extension – fixing an edge used for the last update as strategy for player one – is not sound: Figure 2.3 shows a small example of a singleton Büchi game, where all positions are positions of player 1 ($V_0 = \emptyset$). The positions a, b, and c are colored by 1, while position d is colored by 2. Player 1 can choose a self-loop at position a (in which case she wins), or move in a Hamiltonian circle (in which case she loses). If we start with twice lifting at position a ($\rho_1(a) = 1$, $\rho_2(a) = 2$) followed by lifting at position b ($\rho_3(b) = 3$), c ($\rho_4(c) = \top$), d ($\rho_5(d) = \top$), and again at a ($\rho_6(a) = \top$) and b ($\rho_7(b) = \top$), all positions are correctly marked as winning for player 1; but the last update of position a relies on $\rho_5(d) = \top$, and the naive approach would result in a losing strategy.

We show that a variance of the algorithm can be used to also construct a winning strategy of player 1 on her complete winning region. It suffices to store intermediate strategies for player 1, and to keep two sets of liftable positions instead of one – one set for positions that are liftable without changing the intermediate strategy of player 1, and one set of positions that are liftable, but only if the strategy of player 1 is changed. The adapted algorithm always gives preference to liftable positions from the first set. If only liftable positions from the latter set remain, one of these positions is lifted and the intermediate strategy is updated accordingly.

In the singleton game from the example of Figure 2.3, we can either start with the self-loop at position a and thus with a winning strategy, or with the loosing strategy to move from a to d. In the first case, we never have to adjust the strategy. (One possible sequence of progress measure updates is $(\rho_1(a) = 1$, $\rho_2(a) = 2$, $\rho_3(a) = 3$, $\rho_4(a) = \top$, $\rho_5(b) = \top$, $\rho_6(c) = \top$, $\rho_7(d) = \top$.) In the latter case, we first compute the fixed point for the singleton game, where the moves of player 1 are restricted by her strategy. (One possible sequence of progress measure updates is $(\rho_1(a) = 1$, $\rho_2(b) = 2$, $\rho_3(c) = 3$.) Once the fixed point for this strategy is reached, the strategy is adjusted by choosing the self-loop at position a. (One possible sequence of further progress measure updates is $(\rho_4(a) = 2$, $\rho_5(a) = 3$, $\rho_6(a) = \top$, $\rho_7(b) = \top$, $\rho_8(c) = \top$, $\rho_9(d) = \top$.)

Theorem 2.9 *For parity games with maximal color 2, the proposed algorithm can be used to solve the parity game and to construct winning strategies for both players in time $O(m\,n)$.*

Proof: The proposed changes to Jurdziński's algorithm only impose a particular order on the lifting operations, which could coincidentally occur in his algorithm, too. This implies the correctness of the least fixed point and thus the correctness of the resulting winning regions and strategy of player 0 (cf. Corollary 2.8).

For the correctness of the winning strategy of player 1 on her winning region, we show by induction that every time the intermediate strategy needs to be changed, say from f to f', the intermediate progress measure ρ_μ^f is the $\sqsubseteq$-least valid 0 progress measure ρ_μ^f for $\mathcal{P}_f$.

Induction Basis: For any initial strategy f the claim holds trivially – up to the first adjustment of the intermediate strategy the algorithm resembles the original algorithm for $\mathcal{P}_f$.

Induction Step: Consider the situation after changing the intermediate strategy from f to f' by choosing a lift-edge (p, p'). Let us compare the $\sqsubseteq$-least valid

0 progress measure ρ_μ^f for $\mathcal{P}_f$ with the $\sqsubseteq$-least valid 0 progress measure $\rho_\mu^{f'}$ for $\mathcal{P}_{f'}$.

First, we have $\rho_\mu^f(p) \neq \rho_\mu^{f'}(p)$ (because $\rho_\mu^f(p) = \rho_\mu^{f'}(p)$ implies $\rho_\mu^f = \rho_\mu^{f'}$), and $\rho_\mu^f(p) \neq \bot$. Let us assume $\rho_\mu^f(p) > \rho_\mu^{f'}(p)$, and choose $\delta = \rho_\mu^f(p) - \rho_\mu^{f'}(p)$. This implies $\rho_\mu^f(q) \geq \rho_\mu^{f'}(q)$ and $\rho_\mu^f(q) - \rho_\mu^{f'}(q) \leq \delta$ for all positions $q \in V$. In particular, this implies $\rho_\mu^f(p') - \rho_\mu^{f'}(p') \leq \delta$, which contradicts the assumption that (p, p') is a lift-edge. $\lightning$

Hence, $\rho_\mu^{f'}(p) > \rho_\mu^f(p)$ holds true, which implies $\rho_\mu^f(q) \geq \rho_\mu^{f'}(q)$ for all positions $q \in V$. □

2.4.2 The Approximation

An essential step in the proposed algorithm is the construction of σ/π-paradises. For their construction, we draw from the efficient computation of the $\sqsubseteq$-least valid σ progress measure (Theorem 2.6).

Instead of using the maximal codomain $\mathcal{M}_\infty^\sigma$, the smaller codomain $\mathcal{M}_\pi^\sigma$ is used for the progress measures, which contains only those functions f that satisfy $\sum_{c=0}^{d} f(c) \leq \pi$ for some parameter $\pi \in \mathbb{N}$. (d denotes the highest color of the parity game). The size of $\mathcal{M}_\pi^\sigma$ can be estimated by

$$|\mathcal{M}_\pi^\sigma| \leq \binom{\pi + \lceil 0.5(d+1) \rceil}{\pi} + 1.$$

Using $\mathcal{M}_\pi^\sigma$ instead of $\mathcal{M}_\infty^\sigma$, $\|\rho_\mu\|$ contains all σ-paradises of size $\leq \pi + 1$ (where ρ_μ denotes the $\sqsubseteq$-least valid σ progress measures).

Theorem 2.10 *Let $\mathcal{P} = (V_0, V_1, E, \alpha)$ be a parity game, and let $P_\sigma \subseteq V$ be a σ-paradise of size $|P_\sigma| \leq \pi + 1$. Then there is a valid σ progress measure $\rho : V \to \mathcal{M}_\pi^\sigma$ with $P = \|\rho\|$.*

Proof: Since P_σ is a σ-paradise, E and $V_{1-\sigma} \cap P_\sigma \times V \setminus P_\sigma$ are disjoint, and player σ has a memoryless strategy f that is winning on every game position in P_σ such that $f(p) \in P_\sigma$ for all $p \in V_\sigma \cap P_\sigma$. If we restrict $\mathcal{P}$ to $\mathcal{P}' = \mathcal{P}_f \cap P_\sigma$, then the winning region of player σ covers the whole set P_σ of game positions of $\mathcal{P}'$. To solve $\mathcal{P}'$, we can use the maximal codomain $\mathcal{M}_\infty^{\sigma\,\prime}$. By Theorem 2.7, the $\sqsubseteq'$-least progress measure ρ'_μ for this codomain satisfies $\|\rho'_\mu\| = P_\sigma$. Since $\mathcal{M}_\infty^{\sigma\,\prime} \subseteq \mathcal{M}_\pi^\sigma$ is contained in $\mathcal{M}_\pi^\sigma$ (P_σ must contain at least one position with even color if $\sigma = 0$, or one position with odd color if $\sigma = 1$, respectively), we can extend ρ'_μ to a valid σ progress measure ρ on $\mathcal{P}$ by setting $\rho(p) = \rho'_\mu(p)$ for all $p \in P_\sigma$, and $\rho(p) = \top$ otherwise. □

By Theorem 2.6, we can compute the $\sqsubseteq$-least valid σ progress measure ρ_μ in time $O(c\,m\,|\mathcal{M}_\pi^\sigma|)$, and, by Theorem 2.5, we can construct a winning strategy for player σ on $\|\rho_\mu\|$ within the same complexity bound.

Corollary 2.11 *For a given parity game $\mathcal{P}$ with c colors and m edges, we can construct a $\sigma/(\pi+1)$-paradise $P_\sigma^{\pi+1}$ for player σ in time $O\big(c\,m\,\binom{\pi+\lceil 0.5c\rceil}{\pi}\big)$. A winning strategy for player σ on $P_\sigma^{\pi+1}$ can be constructed within the same complexity bound.* □

2.4.3 Correctness

In this subsection, we show that Procedure *Winning-Regions* computes the winning regions correctly.

Theorem 2.12 *For a given parity game $\mathcal{P}$, Procedure* Winning-Regions *computes the complete winning regions of both players.*

Proof: We prove the claim by induction. Let d denote the highest color of $\mathcal{P}$.
Induction Basis ($d \leq 2$): For $d \leq 2$, the algorithm follows the approach introduced in Subsection 2.4.1 (c.f. Theorem 2.9).
Induction Step: By induction hypothesis, the procedure works correctly for all parity games with highest color less than or equal to $d-1$. Let $\mathcal{P}$ be a parity game with highest color d, and let $\sigma = d \mod 2$.

The call of the Procedure *Approximate* in line 6*a* provides a (possibly empty) $(1-\sigma)/\pi$-paradise (Theorem 2.10). The $(1-\sigma)$-attractor of this set is then added to the winning region of $1-\sigma$ (line 6*b*), and subtracted from $\mathcal{P}$ (line 6*c*), which is safe by Lemma 2.3.

In line 6*d*, the σ-attractor A of the set of states with color d is subtracted from $\mathcal{P}$, and the resulting parity game $\mathcal{P}' = \mathcal{P} \setminus A$ is solved by recursively calling the Procedure *Winning-Regions* (line 6*e*). Since the highest color of $\mathcal{P}'$ is strictly smaller than d, the resulting winning regions are correct. The winning region $W'_{1-\sigma}$ of player $1-\sigma$ is a $(1-\sigma)$-paradise in $\mathcal{P}'$, and, due to the σ-attractor construction, also in $\mathcal{P}$. If $W'_{1-\sigma}$ is non-empty, then the $(1-\sigma)$-attractor of this set is added to the winning region of player $1-\sigma$ (line 6*g*), and subtracted from $\mathcal{P}$ (line 6*h*), which is safe by Lemma 2.3.

Since the size of $\mathcal{P}$ is strictly reduced in every iteration of the loop, the set $W'_{1-\sigma}$ returned after the recursive call in line 6*e* is eventually empty, and the procedure terminates. When $W'_{1-\sigma}$ is empty, player σ wins from all positions in (the remaining) parity game $\mathcal{P}$ by following a memoryless strategy that agrees

on every position in $\mathcal{P}'$ with a memoryless winning strategy f on $\mathcal{P}'$, makes an arbitrary (but fixed) choice for positions with color d, and follows an attractor strategy (from the σ-attractor construction of line 6d) on the remaining positions.

An f-conform play is winning for player σ by Lemma 2.4. □

Note that the all operations can be extended to also return the winning strategies for both players without extra cost.

2.4.4 Complexity

While the correctness of the algorithm is independent of the chosen parameter, its complexity crucially depends on this choice. This subparagraph covers two cases: Parity games with a reasonable number of colors, and parity games with a high number of colors.

Parameter for $c \in O(\sqrt{n})$. For the important class of parity games with a reasonable number of colors – $c \in O(\sqrt{n})$ – we choose the parameter such that the cost for the recursive call (line 6e) coincides with the complexity of computing the approximation (line 6a). First, we show that the Procedure *Winning-Regions* indeed proceeds in big steps.

Lemma 2.13 *For a parameter $\pi(n, c)$, the repeat loop of the algorithm is iterated at most $\left\lfloor \frac{n}{\pi(n,c)+2} \right\rfloor + 1$ times.*

Proof: As discussed in the proof of Theorem 2.12, the $(1-\sigma)$-attractor A of the computed approximation $P^{\pi}_{1-\sigma}$ (line 6a) and the winning region $W'_{1-\sigma}$ of player $1-\sigma$ are $(1-\sigma)$-paradises on $\mathcal{P}$ and $\mathcal{P} \setminus A$, respectively. Thus, their union U is a $(1-\sigma)$-paradise on $\mathcal{P}$. If the size of U does not exceed $\pi + 1$, U is contained in $P^{\pi}_{1-\sigma}$ by Corollary 2.11. In this case, $W'_{1-\sigma}$ is empty, and the loop terminates. Otherwise, a superset of U is subtracted from P during the iteration (lines 6c and 7h), which can happen at most $\left\lfloor \frac{n}{\pi(n,c)+2} \right\rfloor$ times. □

Building on this lemma, it is simple to define the parameter π such that the requirement of equal complexities is satisfied. We fix the function γ such that $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{\lceil 0.5c \rceil \lfloor 0.5c \rfloor}$ if c is odd, and $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lceil 0.5c \rceil \lfloor 0.5c \rfloor}$ if c is even, and choose $\beta(c) = \frac{\gamma(c)}{\lfloor 0.5c \rfloor + 1}$. These definitions imply $\gamma(c+1) = \gamma(c) + 1 - \beta(c)$.

Theorem 2.14 *Solving a parity game $\mathcal{P}$ with $c > 2$ colors, m edges, and n game positions can be performed in time $O\big(m \left(\frac{\kappa\, n}{c}\right)^{\gamma(c)}\big)$. ($\kappa$ is a small constant.)*

Proof: We prove, equivalently[2], that solving a parity game with highest color d takes at most $\kappa_1 \frac{m}{\sqrt[3]{d!}}(\kappa_2 n)^{\gamma(d+1)} + \kappa_3\, m$ steps.

We choose the parameter $\pi(n,d) = \lceil 2\sqrt[3]{d}\, n^{\beta(d)} \rceil$, and first determine the complexity of running the approximation procedure. By Corollary 2.11, the running time of the approximation procedure can be estimated by $\kappa_4\, c\, m \left(\begin{smallmatrix} \pi(n,d) + \lceil 0.5c \rceil \\ \pi(n,d) \end{smallmatrix} \right)$ steps.

With the chosen parameter, $(\pi(n,d) + \lceil 0.5c \rceil)^{\lceil 0.5c \rceil}$ can be estimated by $\kappa_5 \big(\kappa_6\, \pi(n,d)\big)^{\lceil 0.5c \rceil}$, and $\frac{(\sqrt[3]{d})^{\lceil 0.5c \rceil}}{\lceil 0.5c \rceil !}$ can be estimated by $\kappa_7 \frac{{\kappa_8}^c}{\sqrt[3]{(d-1)!}}$.

With $\gamma(d) = \lceil 0.5c \rceil \beta(d)$, this results in an estimation of $\kappa_1 \frac{m}{\sqrt[3]{(d-1)!}}(\kappa_2 n)^{\gamma(d)}$ steps for the running time of the approximation procedure.

Induction Basis ($d \leq 2$): The complexity $\mathcal{O}(m\, n) = O(m\, n^{\gamma(3)})$ (cf. Theorem 2.9) of solving a three color game does not depend on the parameter.

Induction Step: By induction hypothesis, the claim holds true for parity games with highest color smaller than d. The time complexity of each iteration of the loop is dominated by the time complexity of the approximation in line 6a, and the complexity of the recursive call in line 6e.

By induction hypothesis, the recursive call takes at most $\kappa_1 \frac{m}{\sqrt[3]{(d-1)!}}(\kappa_2 n)^{\gamma(d)} + \kappa_3\, m$ steps. We showed that the approximation can be computed in $\kappa_1 \frac{m}{\sqrt[3]{(d-1)!}}(\kappa_2 n)^{\gamma(d)}$ steps, and, finally, the joint running time for all other operations in (and prior to) the loop can be estimated by $\kappa_3\, m$.

By Lemma 2.13, we can estimate the number of iterations of the loop by $\left\lceil \frac{n^{1-\beta(d+1)}}{2\sqrt[3]{d}} \right\rceil$ iterations, and with $\gamma(d+1) = \gamma(d) + 1 - \beta(d)$ we obtain the claimed complexity bound. □

This coarse estimation already shows that we can choose any value higher than 1, $\sqrt[6]{e}$, $2\sqrt{2e}$, and $(2e)^{1.5}$ for κ_6, κ_8, κ_2, and κ, respectively.

Parameter for $c \in \omega(\sqrt{n})$. For this class of parity games, we choose the parameter π such that the size of the call tree equals the time complexity of computing the $(1-\sigma)/\pi$-paradise (line 6e). The results of Jurdziński, Zwick and Paterson [JPZ06] imply that choosing the parameter $\pi = \sqrt{n}$ results in a call tree of size $n^{O(\sqrt{n})}$. The cost between two calls is dominated by the cost for

[2]The occurring κ_i are suitable small constants. The proof makes use of $(e^{-1}d)^d \leq d!$ and $d!$ in $O((\kappa' d)^d)$ for all $\kappa' > e^{-1}$, and subsumes polynomial occurrences of d in the respective constant κ_i. Note that $d = c - 1$ and $\lceil 0.5c \rceil = \lfloor 0.5d \rfloor + 1$.

computing the $(1-\sigma)/\pi$-paradise, which is approximately $\binom{\pi+n}{\pi}$, and thus in $n^{O(\sqrt{n})}$.

Corollary 2.15 *Solving a parity game $\mathcal{P}$ with n game positions can be performed in time $n^{O(\sqrt{n})}$.*

2.5 Discussion

In this chapter, a novel approach to solving parity games has been proposed that reduces the complexity bound for solving parity games from $O\big(c\,m\,(\frac{2n}{c})^{\lfloor 0.5\,c\rfloor}\big)$ for the qualitative analysis of parity games – and $O\big(c\,m\,(\frac{2n}{c})^{\lceil 0.5\,c\rceil}\big)$ for the construction of winning strategies for both players [Jur00] – to $O\big(m\,\big(\frac{\kappa\,n}{c}\big)^{\gamma(c)}\big)$ for $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lceil \frac{c}{2}\rceil \lfloor \frac{c}{2}\rfloor}$ if c is even, and $\gamma(c) = \frac{c}{3} + \frac{1}{2} - \frac{1}{\lceil \frac{c}{2}\rceil \lfloor \frac{c}{2}\rfloor}$ if c is odd (where κ is a small constant).

This reduces the exponential factor from $\lfloor \frac{c}{2}\rfloor$ and $\lceil \frac{c}{2}\rceil$, respectively, to less than $\frac{c}{3} + \frac{1}{2}$. After the reduction from $c-1$ [McN93, EL86, Zie98] to $\lceil \frac{c}{2}\rceil + 1$ by Browne et al. [BCJ+97], this is the second reduction that reduces the exponential growth with the number of colors in the long history of algorithms for solving parity games.

The improved complexity bound also raises a new question. While solving parity games with a fixed number of colors is known to be in **P**, the degree of the polynomial increases with the number of colors. For McNaughton's algorithm, for example, the degree increases by 1 with every additional color, and in Jurdziński's algorithm the degree increases by 1 for every even color if we are only interested in the winning regions, and by 1 for every odd color if we also require the winning strategies as witnesses. Different to both approaches, there is no increase in the complexity when moving from 2 to 3 colors in the proposed big step algorithm. This anomaly suggests that either the complexity of solving Büchi games can be improved, or that there is still some leeway for improving the complexity of solving parity games. This is particularly interesting in the light of the wide open complexity of the optimal strategy improvement algorithm proposed in the following chapter: The best lower bound for the complexity of that algorithm with an arbitrary number of colors is close to the known complexity bounds for solving 2 and 3 color games.

Chapter 3

An Optimal Strategy Improvement Method for Solving Parity Games

Abstract

This chapter proposes a strategy improvement algorithm for parity games, which is guaranteed to select, in each improvement step, an optimal combination of local strategy modifications. Current strategy improvement methods stepwise improve the strategy of one player with respect to some ranking function, using an algorithm with two distinct phases: They first choose a modification to the strategy of one player from a list of *locally* profitable changes, and subsequently evaluate the modified strategy. This separation is unfortunate, because current algorithms have no effective means to predict the *global effect* of the individual local modifications beyond classifying them as profitable, adversarial, or stale. Furthermore, they are completely blind towards the *cross effect* of different modifications: Applying one profitable modification may render all other profitable modifications adversarial. Our new construction overcomes the traditional separation between choosing and evaluating the modification to the strategy. It thus improves over current strategy improvement algorithms by providing the *optimal improvement* in every step, selecting the best combination of local updates from a superset of all profitable and stale changes.

3.1 Introduction

The traditional forward techniques [Jur00] for solving parity games ($\approx O(m\, n^{\frac{1}{2}c})$ for games with n positions, m edges and c colors), backward techniques [McN93, EL86, Zie98] ($\approx O(m\, n^c)$), and their combination ($\approx O(m\, n^{\frac{1}{3}c})$, cf. Chapter 2) aim at good complexity bounds. However, these bounds are sharp. Moreover, forward techniques [Jur00] are likely to display their worst case complexity on most practical examples, and the combined forward-backward algorithm discussed in Chapter 2 inherits this disadvantage.

While this line of research seems to be exhausted, strategy improvement algorithms [Lud95, Pur95, VJ00, BV07] for parity and payoff games are still in their infancy. In this chapter, an efficient strategy improvement algorithm is introduced, shifting the aim from provably good complexity bounds to efficient algorithms for solving parity games that perform well in practice.

State of the Art. Strategy improvement algorithms are simplex style algorithms that are closely related to the simplex algorithm for solving linear programming problems. Strategy improvement methods assign a value to each infinite play of a parity or payoff game, and the objective of the two participating players (called player 0 and player 1) is to minimize and maximize this value, respectively.

In strategy improvement algorithms for parity and payoff games [Lud95, Pur95, VJ00, BV07], the memoryless strategies of player 0 define the corners of a simplex. For each memoryless strategies of player 0, her opponent has an optimal memoryless counter strategy. This pair of strategies defines a pointwise ranking function that assigns to each game position p the value (or rank) of the play that starts in p.

The two distinguishing differences between strategy improvement techniques compared to the simplex algorithm are a *weak pivot rule* and the option of *multiple modifications* in every step.

Weak Pivot Rule. Different to simplex techniques for linear programming problems, current strategy improvement methods do not take the *global effect* of a step to an adjacent corner of the simplex into account. When estimating the improvement obtained by a local modification to a strategy, they presume that changing the strategy for one game position has no influence on the rank of any other game positions. For example, in the situation depicted in Figure 3.1a, player 0 can choose between two improvements of her strategy from her position colored by 1; she can either move to the position with color 2, or to the

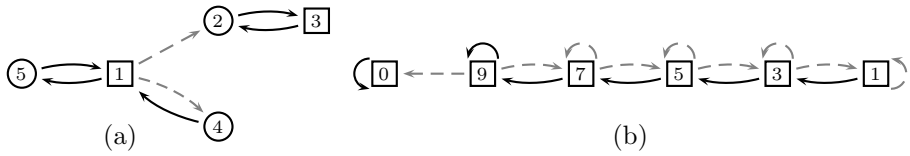


Figure 3.1: The examples show situations where ignoring global effects (a) and cross effects between different updates (b) perturb the pivot rule of current strategy improvement methods. States of player 0 and player 1 are depicted as boxes and circles, respectively. The current strategy of player 0 (and all options of player 1) are depicted as full arrows, the alternative moves of player 0 are represented by dashed arrows.

position with color 4. While the latter choice is obviously better (because player 0 asserts the parity condition), the local analysis considers only the value of the positions for the *old* strategy [Lud95, Pur95, VJ00, BV07]. A valuation function based on the old strategy, however, will favor the change to the position with color 2, because the dominating color in the infinity set for this position (for the old strategy) is 3, while the dominating color in the infinity set of the position colored by 4 is 5. In the whole, the local analysis alone does not provide much more information than a classification into locally profitable, adversarial, and stale modifications.

Multiple Modifications. An advantage of strategy improvement methods over the simplex method for linear programming is the option to consider several locally profitable modifications at the same time [Pur95, VJ00]. This advantage, however, must be considered with care, because current strategy improvement methods are blind towards the *cross effect* between different local updates. (The cross effect of different modifications is harder to predict than the global effect of a single modification.)

While any combination of profitable changes remains profitable, it may happen that applying one modification turns all remaining modifications adversarial. In the small singleton parity game depicted in Figure 3.1b, player 0 is only one step away from her optimal strategy. (It suffices to update the strategy in the position with color 9.) All local changes lead to an improvement, but after updating the strategy at the position with color 9, all remaining changes become harmful. Given this shortcoming, it is unclear whether or not simultaneous updates are a step forward for current strategy improvement algorithms.

Current Strategy Improvement Algs	Optimal Strategy Improvement Algorithm
1. pick initial strategy	1. pick and evaluate initial strategy
2. evaluate current strategy	2. adjust evaluation, increasing #profitable/stale mods
3. chose from profitable modifications	3. find and evaluate optimal combination of p/s mods
4. goto 2	4. goto 2

Figure 3.2: Comparison between traditional strategy improvement methods and the proposed optimal improvement algorithm. While current techniques first choose a particular update from profitable modifications and subsequently evaluate it, our novel technique concurrently considers all combinations of profitable and stale modifications.

Contribution. We introduce a strategy improvement algorithm that is based on a reduction to simple update *games*, which can be solved in a single sweep. It provides substantial advantages over current strategy improvement algorithms:

1. *The reduction is more natural.* It reduces solving parity (or mean payoff) games to solving a series of simplified games, where the options of player 0 are restricted, but not to the extreme of a singleton game. It thus preserves the game character of the problem rather than treating it as a purely graph theoretic problem.

2. *The improvements are greater.* The game-theoretic approach allows us to take the global and cross effects of different local modifications to the strategy into account. We thus overcome the critical blind spot of current strategy improvement algorithms and can, for the first time, make full use of the strength attached to simultaneous modifications.

3. *The game theoretic analysis is cheaper.* Reductions to graph theoretic algorithms need to exclude stale cycles. Both, for parity and payoff games, the codomain of the pointwise ranking function needs to be increased by a factor linear in the size n of the game, which raises the estimation for the amount of iterations by a factor of n and slows down the arithmetic operation.

From Graph-based to Game-based Updates. The suggested optimal strategy improvement algorithm reduces solving parity games to solving a series of simpler two player games. Turning to a game theoretic (rather than to a graph theoretic) approach allows for considering *all combinations* of profitable and stale modifications in every update step, taking all global and cross effects into account.

This advancement is achieved by a novel technique that resolves the separation between choosing and evaluating the modifications to a strategy. Where current strategy improvement algorithms first update the strategy and then evaluate the resulting singleton game, our approach exploits a natural preorder for the evaluation of game positions that allows for simultaneously constructing optimal strategies for both players, such that every game position is only considered once. Following this preorder, the evaluation of each individual position can be reduced to a cheap local analysis.

The intuition for the preorder is that, in most cases, the game-theoretic approach allows for fixing an optimal decision for every position *after* all of its successors have been reevaluated. If all positions do have unevaluated successors, we can immediately determine the optimal choice for some position of player 1.

A side-effect of the game theoretic approach is that staling becomes our confederate. All current strategy improvement methods, on the other hand, build on some guarantee that stale modifications can safely be ignored (there, only profitable changes can be processed). This is often bought by blowing up the statespace [VJ00, BV07]. In the game-based approach, stale changes increase the number of updates taken into account and thus strengthen our global pivot rule.

The Ranking Function. We change the rules of parity games by allowing one player, say player 0, to terminate the game in her positions. This is related to the finite unraveling of mean payoff games suggested by Zwick and Paterson [ZP96] and the controlled single source shortest path problem from the reduction of Björklund and Vorobyov [BV07].

The objective of player 0 remains to assert an infinite path with even maximal priority in the infinity set. However, we add the natural secondary objective for the case that she has not yet found (or there is no) such strategy. If player 0 cannot assert such a path, she eventually stops the unraveling of the game, optimizing the finite occurrences of the different priorities, but *disregarding* the number of positions with priority 0. (Disregarding this number leads to a coarser ranking function and to an improved estimation of the number of improvement steps. It also leads to greater improvements by increasing the number of profitable or stale modifications.) Second, if the highest occurring priority is odd, there is no need to keep track of the number of occurrences of this priority. It suffices to store the information that this maximal number occurs on a finite path, resulting again in a coarser ranking function.

For parity games with n positions, m edges, and c colors, the coarser ranking function leads to an improved estimation of the number of updates from the currently best bound $O\big(n\,(\frac{n+c}{c})^{c+1}\big)$ [BV07] for the number of arithmetic operations needed by strategy improvement algorithms to $O\big(n\,(\frac{n+c}{c})^{c-1}\big)$ for parity games with an even number of colors, and to $O\big(n\,(\frac{n+c}{c})^{c}\big)$ if the numbers of colors is odd, reducing the bound by a factor quadratic and linear in the number of states, respectively.

The complexity of the strategy improvement algorithm proposed in this chapter is therefore good enough for the complexity results for the constructive non-emptiness test for deterministic and nondeterministic parity tree automata in Chapter 4 and its applications to satisfiability and synthesis problems in Parts II and III).

3.2 Escape Games

Escape games are total reward games that are tailored for the optimal improvement method. They generalize parity games by allowing player 0 to terminate every play immediately on each of her positions. Technically this is done by extending the arena with a fresh *escape position*, which forms a sink of the extended arena, and can be reached from every position of player 0. Every play of an *escape game* either eventually reaches the escape position and then terminates, or it is an infinite play in the non-extended arena.

Bipartite Arena. In this chapter, we assume the arenas to be bipartite for technical convenience. The position in the construction where this assumption is used is marked explicitly, and the extension to games with general arenas is described.

Extended Arena. In an escape game, the finite arena $\mathcal{A} = (V_0, V_1, E)$ is extended to the directed graph $\mathcal{A}' = (V_0, V_1', E')$, which extends the arena $\mathcal{A}$ by a fresh position $\bot$ of player 1 ($V_1' = V_1 \uplus \{\bot\}$) that is reachable from every position of player 0 ($E' = E \cup V_0 \times \{\bot\}$). The escape position is a sink in $\mathcal{A}'$.

Finite Plays. Since the escape position is a sink, every play terminates when reaching $\bot$. The set of plays is therefore extended by the finite plays $\pi = p_0 p_1 p_2 p_3 \ldots \bot \in (V_0 \uplus V_1)^* \{\bot\}$.

Escape Games. An *escape game* is a game $\mathcal{E} = (V_0, V_1, E, \alpha)$, where $\mathcal{A} = (V_0, V_1, E)$ is a finite arena, and $\alpha : V_0 \cup V_1 \to \mathcal{C} \subset \mathbb{N}$ is a coloring function. An escape game is played on the extended arena $\mathcal{A}' = (V_0, V_1', E')$.

An infinite play $\pi = p_0 p_1 p_2 \ldots \in V^\omega$ of an escape game is evaluated to ∞ if the highest color occurring infinitely often is even, and to $-\infty$ otherwise. A finite play $\pi = p_0 p_1 p_2 \ldots p_n \perp$ is evaluated to a function $\rho(\pi) : \mathcal{C}_0 \to \mathbb{Z}$ (where $\mathcal{C}_0 = \mathcal{C} \setminus \{0\}$ is the codomain of the coloring function without 0) that maps an element c' of $\mathcal{C}_0$ to the number of positions p_i in π with $i > 0$ that are colored by $c' = \alpha(p_i)$. (Disregarding the color of the first position is technically convenient.)

The potential values of a path are ordered by the obvious alphabetic order $>$ that sets $\rho > \rho'$ if

- the highest color c' with $\rho(c) \neq \rho'(c)$ is even and $\rho(c') > \rho'(c')$, or
- the highest color c' with $\rho(c) \neq \rho'(c)$ is odd and $\rho'(c') > \rho(c')$.

Additionally, we define $\infty > \rho > -\infty$. The objective of player 0 is to maximize this value, while it is the objective of player 1 to minimize it.

We introduce an operator $\oplus$ for the evaluation of finite paths. For $\mathcal{R} = (\mathcal{C}_0 \to \mathbb{Z}) \cup \{\infty\}$, $\oplus : \mathcal{R} \times \mathcal{C} \to \mathcal{R}$ maps a function ρ and a color c' to the function ρ' that deviates from ρ only by assigning the respective successor $\rho'(c') = \rho(c') + 1$ to c' (and leaves $\rho'(d) = \rho(d)$ for $d \neq c'$). We fix $\infty \oplus c' = \infty$ and $\rho \oplus 0 = \rho$.

Estimations. We introduce *estimations* $v : V_0 \cup V_1' \to \mathcal{R}$ for an escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$ as witnesses for the existence of a memoryless strategy f of player 1, which guarantees that every f-conform play π starting in some position p is evaluated to $\rho(\pi) \geq v(p)$. Formally, an estimation v has to satisfy the following side conditions:

- $v(\perp) = \mathbf{0}$ ($\mathbf{0}$ denotes the constant function that maps all colors in $\mathcal{C}_0$ to 0),
- for every position $p \in V_0$ of player 0 there is an edge $e = (p, p') \in E'$ such that $v(p) \leq v(p') \oplus \alpha(p')$ holds true,
- for every position $q \in V_1$ of player 1 and every edge $e = (q, q') \in E$, $v(q) \leq v(q') \oplus \alpha(q')$ holds true, and
- player 0 has a strategy f_∞ that maps every position $p \in V_0$ of player 0 with $v(p) = \infty$ to a position $p' = f_\infty(p)$ with $v(p') = \infty$, and which guarantees that every f_∞-conform play π starting in p is evaluated to $\rho(\pi) = \infty$.

A trivial estimation is simple to construct: We denote with v_0 the estimation that maps the escape position to $v_0(\bot) = \mathbf{0}$, every position $p \in V_0$ of player 0 to $v_0(p) = \mathbf{0}$, and every position $q \in V_1$ to $v_0(q) = min\{\mathbf{0} \oplus \alpha(q') \mid (q, q') \in E\}$.

Remark. The simple construction of the trivial estimation is the only position in this chapter where the restriction to bipartite games is used. For arbitrary parity games, we would first test if player 1 can win from some position without visiting any position of player 0; such positions can (and need to) be removed recursively. Additionally, we had to construct the 'best' path of player 1 to some position of player 0. These constructions are cheap and simple, and would only blur the outline of the algorithm.

Lemma 3.1 *For every estimation v of an escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$ there is a memoryless strategy f for player 0 such that every f-conform play π starting in any position p satisfies $\rho(\pi) \geq v(p)$.*

Proof: We fix an arbitrary strategy f for player 0 that agrees with f_∞ on every position $p \in V_0$ of player 0 with infinite estimation ($v(p) = \infty \Rightarrow f(p) = f_\infty(p)$), and chooses some successor that satisfies $v(p) \leq v\big(f(p)\big) \oplus \alpha\big(f(p)\big)$ otherwise. Every cycle reachable in an f-conform play has non-negative weight (that is, weight $\mathbf{0} \oplus \alpha(p_0) \oplus \ldots \oplus \alpha(p_n)$ of every cycle $p_0 \ldots p_n p_0$ is $\geq \mathbf{0}$) by construction of f; every infinite f-conform play π is therefore evaluated to $\rho(\pi) = \infty \geq v(p)$.

By induction over the length of finite f-conform plays π that start in some position p, we can show that $\rho(\pi) \geq v(p)$. □

For a given an estimation v, the question arises whether this estimation can be improved. We call an estimation v' an *improvement* of an estimation v if $v'(p) \geq v(p)$ holds for all positions $p \in V_0 \cup V_1$, and we call an improvement *strict* if $v' \neq v$.

For every estimation v, we define the *improvement arena* $\mathcal{A}_v = (V_0, V_1, E_v)$ that contains an edge $e = (p, p')$ if it satisfies $v(p) \leq v(p') \oplus \alpha(p')$ (that is, $E_v = \{(p, p') \in E' \mid v(p) \leq v(p') \oplus \alpha(p')\}$; E_v thus contains every edge that originates from a position of player 1), and the 0-*arena* $\mathcal{A}_v^0 = (V_0, V_1, E_v^0)$ which contains an edge $e = (p, p') \in E_v$ of the improvement arena if it satisfies

- $v(p) = v(p') \oplus \alpha(p')$, and
- if e originates from a position $p \in V_0$ of player 0, if additionally no edge $e' = (p, q)$ with $v(p) < v(q) \oplus \alpha(q)$ originates from p

$(E_v^0 = \{(p,p') \in E_v \mid v(p) = v(p') \oplus \alpha(p') \text{ and } p \in V_0 \Rightarrow \forall (p,q) \in E_v . v(p) = v(q) \oplus \alpha(q)\})$.

We call an estimation *improvable* if the 1-attractor of the escape position in the 0-arena $\mathcal{A}_v^0$ does not cover all positions that are not estimated to ∞.

Theorem 3.2 *For every non-improvable estimation v of an escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$ player 1 has a memoryless strategy f' such that every f'-conform play π starting in any position p satisfies $\rho(\pi) \leq v(p)$.*

Proof: We fix a strategy f' for player 1 that agrees on all positions $V_1 \setminus v^{-1}(\infty)$ with some 1-attractor strategy to reach the escape position in the 0-arena $\mathcal{A}_v^0$.

For plays starting in some position p that is evaluated to ∞, $\rho(\pi) \leq v(p) = \infty$ holds trivially. For plays starting in some position p that is not evaluated to ∞, we can show by induction over the length of f'-conform plays starting in p that no f'-conform play can reach a position p' that is evaluated to $v(p') = \infty$. By construction of f', every reachable cycle in an f'-conform play that does not reach a position in $v^{-1}(\infty)$ has negative weight (that is, a weight $< \mathbf{0}$), and every infinite f'-conform play which starts in a position p that is not evaluated to ∞ thus satisfies $-\infty = \rho(\pi) < v(p)$.

For every finite f'-conform play π starting in some position p, we can show by induction over the length of π that $\rho(\pi) \leq v(p)$ holds true. □

The non-improvable estimation of an escape game can be used to defer the winning regions ($v^{-1}(\infty)$ for player 0) and the winning strategy for player 1 on his winning region in the underlying parity game. f_∞ defines the winning strategy of player 0 on her winning region.

3.3 Solving Escape Games

In this section we introduce an optimal strategy improvement algorithm for the fast improvement of estimations for escape games. Every estimation (for example, the trivial estimation v_0) can be used as a starting point for the algorithm.

3.3.1 Optimal Improvement

The estimations that we construct intuitively refer to strategies of player 0 for the extended arena. (Although estimations are a more general concept; not all estimations refer to a strategy.) The edges of the improvement arena $\mathcal{A}_v$ of an escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$ and an estimation v that originate in positions of

player 0 refer to all promising strategy updates, that is, all strategy modifications that *locally* lead to a – not necessarily strict – improvement (profitable and stale modifications). We call an improvement v' of v *optimal* if it dominates all other estimations $\widehat{v}$ that refer to (memoryless) strategies of player 0 that contain only improvement edges. Finding this optimal improvement thus relates to solving an *update game*, which deviates from the full escape game $\mathcal{E}$ only by restricting the choices of player 0 to her improvement edges.

3.3.2 Basic Update Step

Instead of computing the optimal improvement v' of an estimation v directly, we compute the optimal update $u = v' - v$. (The operator $+ : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ maps a pair ρ, ρ' of functions to the function ρ'' that satisfies $\rho''(c') = \rho(c') + \rho'(c')$ for all $c' \in \mathcal{C}_0$. $-$ is defined accordingly.)

For a given escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$ with estimation v, we define the *improvement potential* of an edge $e = (p, p') \in E_v$ in the improvement arena $\mathcal{A}_v$ as the value $P(e) = v(p') \oplus \alpha(p') - v(p) \geq \mathbf{0}$ by which the estimation would locally be improved when the respective player chose to turn to p' (disregarding the positive global effect that this improvement may have). To construct the optimal update, we construct the improvement arena, and evaluate the optimal update of the escape position to $u(\bot) = 0$. We then evaluate the improvement of the remaining positions successively by applying the following evaluation rule:

1. if there is a position $p \in V_1$ of player 1 that has only evaluated successors, we evaluate the improvement of p to $u(p) = min\{u(p') + P\big((p, p')\big) \mid (p, p') \in E\}$,

2. else if there is a position $p \in V_1$ of player 1 that has an evaluated successor p' with $u(p') = P\big((p, p')\big) = 0$, we evaluate the improvement of p to $u(p) = 0$,

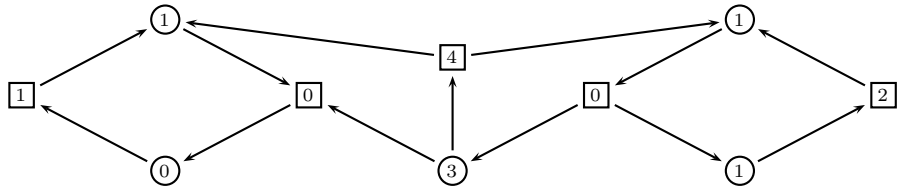
3. else if there is a position $p \in V_0$ of player 0 that has only evaluated successors, we evaluate its improvement to $u(p) = max\{u(p') + P\big((p, p')\big) \mid (p, p') \in E_v\}$[1],

4. else we choose a position $p \in V_1$ of player 1 with minimal intermediate improvement $u'(p) = min\{u(p') + P\big((p, p')\big) \mid p'$ is evaluated and $(p, p') \in E\}$ and evaluate the improvement of p to $u(p) = u'(p)$. (Note that $min\{\emptyset\} = \infty$.)

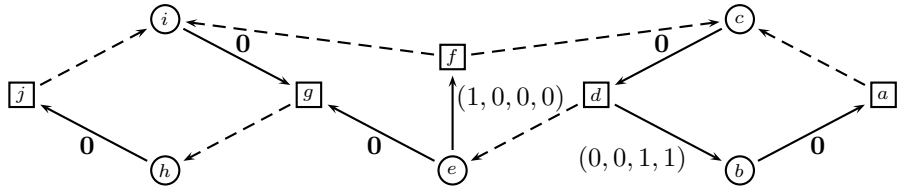
[1] We could also choose $u(p) = max\{u(p') + P\big((p, p')\big) \mid p'$ is evaluated and $(p, p') \in E\}$.

3.3.3 Illustrating Example

The basic update step is illustrated by the following example.



The positions of player 0 are denoted as boxes, and the positions of player 1 are denoted as circles. The positions are decorated with their color. Solving this parity game reduces to solving the following escape game; in order to keep the representation simple, the escape position is not depicted. The full edges are improvement edges in $\mathcal{A}_{v_0}$ (where v_0 is the trivial estimation), they are decorated with their improvement potential. The positions are marked with an identifier.



The following table shows the development of the estimations, starting with the trivial estimation v_0. v_1 and v_2 are the estimations computed after the first and second application of the basic update step, respectively.

position	player	v_0	v_1	v_2
a	0	**0**	**0**	∞
b	1	$(0,0,1,0)$	$(0,0,1,0)$	∞
c	1	**0**	$(0,0,1,1)$	∞
d	0	**0**	$(0,0,1,1)$	∞
e	1	**0**	**0**	**0**
f	0	**0**	**0**	∞
g	0	**0**	**0**	**0**
h	1	$(0,0,0,1)$	$(0,0,0,1)$	$(0,0,0,1)$
i	1	**0**	**0**	**0**
j	0	**0**	**0**	**0**

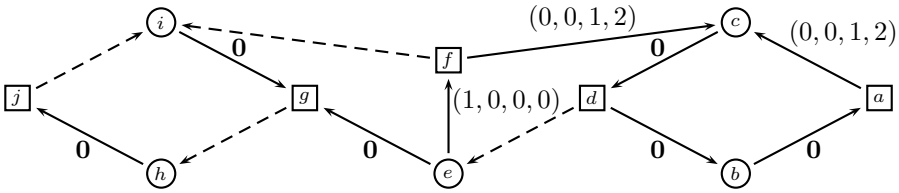
v_2 turns out to be non-improvable, and the algorithm terminates after the next basic update step.

First Basic Upsdate Step. After having determined the improvement potential of the edges and the set of improvement edges, we construct the improvement arena, and run the algorithm. One possible run would be:

1. $u(\bot)$ is set to $\mathbf{0}$
2. rule 3 is applied to position a — $u(a) = \mathbf{0}$
3. rule 1 is applied to positions b — $u(b) = \mathbf{0}$
4. rule 3 is applied to position d — $u(d) = (0, 0, 1, 1)$
5. rule 1 is applied to position c — $u(c) = (0, 0, 1, 1)$
6. rule 3 is applied to position f — $u(f) = \mathbf{0}$
7. rule 3 is applied to position j — $u(j) = \mathbf{0}$
8. rule 1 is applied to position h — $u(h) = \mathbf{0}$
9. rule 3 is applied to position g — $u(g) = \mathbf{0}$
10. rule 1 is applied to position e — $u(e) = \mathbf{0}$
11. rule 1 is applied to position i — $u(i) = \mathbf{0}$

v_1 is then set to $v_1 = v_0 + u$.

Second Basic Update Step. Again, we first determine the improvement potential of the edges, and the set of improvement edges.



We then construct the improvement arena, and run the algorithm. One possible run would be:

1. $u(\bot)$ is set to $\mathbf{0}$
2. rule 3 is applied to position j — $u(j) = \mathbf{0}$
3. rule 1 is applied to position h — $u(h) = \mathbf{0}$
4. rule 3 is applied to position g — $u(g) = \mathbf{0}$
5. rule 1 is applied to position i — $u(i) = \mathbf{0}$
6. rule 2 is applied to position e — $u(e) = \mathbf{0}$
7. rule 4 is applied to position b — $u(b) = \infty$
8. rule 3 is applied to position d — $u(d) = \infty$
9. rule 1 is applied to position c — $u(c) = \infty$
10. rule 3 is applied to position a — $u(a) = \infty$
11. rule 3 is applied to position f — $u(f) = \infty$

v_2 is then set to $v_2 = v_1 + u$. Performing a third basic update step would show that the fixed point is reached.

3.3.4 Correctness

The basic intuition for the optimal improvement algorithm is to re-estimate the value of a position only *after* all its successors have been re-estimated. In this situation, it is easy to determine the optimal decision for the respective player. In a situation where all unevaluated positions do have a successor, we exploit that every cycle in $\mathcal{A}_v$ has non-negative weight (weight $\geq \mathbf{0}$), and every infinite play in $\mathcal{A}_v$ is evaluated to ∞. An optimal strategy of player 1 will thus turn, for some position of player 1, to an evaluated successor. It is safe to chose a transition such that the minimality criterion on the potential improvement u' is satisfied, because, independent of the choice of player 1, no better potential improvement can arise at any later time during this update step. Following these evaluation rules therefore provides an optimal improvement.

Theorem 3.3 *For every estimation v of an escape game $\mathcal{E} = (V_0, V_1, E, \alpha)$, the algorithm computes the optimal improvement $v' = v + u$. If v is improvable, then the optimal improvement $v' \neq v$ is strictly better than v.*

Proof: During the reevaluation, we can fix optimal strategies f and f' for player 0 and 1, respectively, by fixing $f(p)$ or $f'(p)$, respectively, to be some successor of p that satisfies the respective maximality or minimality requirement. (In rule 2, we implicitly apply the same minimality requirement as in rule 4.)

Every infinite f-conform play is evaluated to ∞, and for every finite f-conform play π that starts in some position p, we can show by induction over the length of π that $\rho(\pi) \geq v'(p)$ holds true.

No f'-conform play $\pi = p_0 p_1 p_2 \ldots$ in $\mathcal{A}_v$ (that is, under the restriction that player 0 can chose only transitions in E_v), which does not start in a position p_0 that is evaluated to ∞, can contain a cycle, because p_{i+1} has been evaluated prior to p_i by construction. Thus, every such f'-conform play in $\mathcal{A}_v$ is finite. For every finite f'-conform play π in $\mathcal{A}_v$ that starts in some position p, we can show by induction over the length of π that $\rho(\pi) \leq v'(p)$ holds true.

It remains to show that the algorithm guarantees progress for improvable estimations. If at least one improvement edge e that originates from a position of player 0 has a positive improvement potential $P(e) > \mathbf{0}$, the claim holds trivially. Let us consider the case that the improvement potential is $P(e) = \mathbf{0}$ for every improvement edge e that originates from a position of player 0. According to the update rules, the algorithm will successively assign $u'(p) = \mathbf{0}$ to all positions in the 1-attractor of $\perp$ in the 0-arena $\mathcal{A}_v^0$. If the attractor covers all positions of $\mathcal{E}$, v is non-improvable by Theorem 3.2. Otherwise, $u'(p) > \mathbf{0}$ holds by definition for every remaining position $p \in V_1$ of player 1 that is not in the 1-attractor of the escape position $\perp$. This implies $u > \mathbf{0}$ and thus $v' = v + u > v$. □

3.3.5 Complexity

In spite of the wide variety of strategies that are considered simultaneously, the update complexity is surprisingly low. The optimal improvement algorithm generalizes Dijkstra's single source shortest path algorithm to two player games. The critical part of the algorithm is to keep track of the intermediate update u', and the complexity of the algorithm depends on the used data structure. The default choice is to use binary trees, resulting in an update complexity of $O(m \log n)$. However, using advanced data structures like 2-3 heaps (cf. [Tak99]) reduces this complexity slightly to $O(m + n \log n)$.

Theorem 3.4 *For an escape game with n positions and m edges, the optimal improvement can be computed using $O(m + \delta \log \delta)$ arithmetic operations, where $\delta \leq n$ denotes the number of positions of player 1 for which the improvement is strict.*

Proof: Let us consider a run of our algorithm that, when applying rule 3, gives preference to updates of positions with improvement 0. Keeping track of these updates is cheap, and giving them preference guarantees that all positions with 0-update are removed before the remainder of the graph is treated.

Let us partition the operations occurring after these 0-updates into

1. operations needed for keeping track of the number of unevaluated successors for positions of player 1 and for finding the direction with maximal improvement for positions of player 0, and

2. all remaining operations.

Obviously, (1) contains only $O(m)$ operations, while the restriction to (2) coincides with a run of Dijkstra's algorithm on a subgraph of the improvement arena. (On the subgraph defined by the strategy f of player 0 referred to in Theorem 3.3.) Dijkstra's algorithm can be implemented to run in $O(m + \delta \log \delta)$ arithmetic operations [Tak99]. □

Theorem 3.5 *The algorithm can be implemented to solve a parity game with n positions, m edges, and c colors in time $O\big(m \left(\frac{n+c}{c}\right)^{c'}\big)$, where $c' = c - 1$ if c is even, and $c' = c$ if c is odd.*

Proof: If both players follow the strategies f and f' from the proof of Theorem 3.3 starting in a position p_0 that is not evaluated to $\infty \neq v'(p_0)$, they reach the escape position $\perp$ on a finite acyclic path $p_0 p_1 \dots p_i \perp$. By induction over the length of this path we can show that $v(p_0) = \mathbf{0} \oplus \alpha(p_i) \oplus \dots \oplus \alpha(p_0)$. Note that, for odd highest color $c - 1$ (and thus for even c), only p_i may be colored by $c - 1$. Thus, the number of updates is, for each position, in $O\big(\big(\frac{n+c}{c}\big)^{c'-1}\big)$.

Let us, for the estimation of the running time, assume that only one small update occurs in every step. 'Only one' leads to a small δ (removing the $\delta \log \delta$ part from the estimation), while 'small update' can be used to reduce the discounted cost for the arithmetic operations on $\mathcal{R}$ to $O(1)$: Before computing the improvement potential P, the update u, and the intermediate update u', we first compute an *abstraction* $a : \mathcal{R} \to \mathbb{Z}$ of these values that maps a function $\rho \in \mathcal{R}$ to 0 if $\rho = \mathbf{0}$, and to $\pm$ the highest integer h with $\rho(h) \neq 0$ otherwise (+ if and only if $r > \mathbf{0}$). Computing the concrete value is then linear in the absolute value of the abstraction (rather than in c). For every edge $e = (p, p')$, updating the improvement potential $a \circ P(e)$ to its new value requires $O(max\{a \circ u(p), a \circ u(p')\})$ steps (using the old u). All other operations on abstract values are in $O(1)$.

To compute u', we proceed in two steps. In a first step, we maintain a 2-3 heap that stores only the abstraction of u', and that contains all positions where u' is above a threshold t that is initialized to $t = 0$. For positions with abstract value t, we keep a 2-3 heap with concrete values for u'. Every time we use rule 4 and find an empty concrete 2-3 heap, we increase t to the minimal value of the abstract 2-3 heap, remove all positions with this abstract value from the abstract heap, and add them (with concrete value) to the concrete heap. The required concrete arithmetic operations are linear in the value of the abstraction $a \circ u(r)$ of the concrete update (rather than in c). In the worst case scenario, 'small updates' implies that the discounted cost of the operations is in $O(1)$. □

3.3.6 Extended Update Step

The basic update step can be improved to an extended update step by three simple and cheap additional computation steps:

1. Recursively remove all positions from $\mathcal{E}$ that have no predecessors, and push them on a solve-me-later stack.

2. Adapt the valuation function v to v' such that the values of positions of player 1 are left unchanged ($v'(p) = v(p) \forall p \in V_1$), and the values of all positions of player 0 are maximally decreased ($v'(p) = max\{v'(p') \ominus \alpha(p) \mid (p', p) \in E\} \forall p \in V_0$). This step again exploits that the game is bipartite.

3. Apply a basic update step.

4. Remove the 0-attractor of all positions that are evaluated to ∞ from $\mathcal{E}$.

The intention of the first step is twofold. The first intention is to simplify the game. Positions without predecessors have no impact on the value of other game positions, and their evaluation can safely be postponed until after the remainder of the game has been evaluated. The second intention is to strengthen the second step. In the second step, we exploit the fact that the basic improvement step benefits from a high number of improvement edges that originate from positions of player 0. This number is increased by changing the estimation v such that the estimation of positions of player 1 remain unchanged, while the estimation of positions of player 0 is decreased. The last step is again used to simplify the game.

An interesting side effect of step 4 is that our game-based improvement algorithm behaves like standard fixed point algorithms [McN93, EL86, Zie98]

on Büchi and Co-Büchi games (parity games with only two colors, without loss of generality 0 and 1). Like in these standard algorithms, we iteratively compute the set of states from which player 0 can stay forever in positions with color 0, and then remove their 0-attractor from the game. The game-based approach described in this section can therefore also be viewed as an alternative generalization of the well accepted algorithms for Büchi and Co-Büchi games to general parity games, which preserves different properties than McNaughton's generalization [McN93, EL86, Zie98].

3.4 Benchmarks and Results

To evaluate the applicability of the optimal strategy improvement algorithm, a prototype of the algorithm was implemented and evaluated on different benchmarks, including random games with and without structure as well as other benchmarks for parity games.

3.4.1 Performance on Random Graphs

A first estimation of the performance of our algorithm on random games showed that the expected number of update games depends mainly on the number of colors and the outdegree, but it seems to be *constant* in the number of positions. Each figure in the tables refer to the *highest* number of iterations observed in 100 samples. (They are therefore incomparable to the average values used in the following subsection.) The first table shows the number of iterations used by the optimal strategy improvement algorithm on parity games with uniformly distributed colors and outdegree six.

positions \ colors	3	4	5	6	7	8	9	10	12	15	20
1.000	2	4	3	4	4	7	5	7	13	13	17
3.000	2	4	3	4	3	6	4	7	8	7	20
10.000	2	4	3	5	3	7	4	7	9	7	11
30.000	2	4	3	5	3	8	4	8	10	6	14
100.000	2	5	3	7	3	8	4	8	10	7	13
300.000	2	5	3	6	3	8	4	9	10	7	14
1.000.000	2	5	3	7	3	8	4	11	11	7	16
3.000.000	2	5	3	7	3	9	4	10	13	7	20
10.000.000	2	5	3	7	3	10	4	11	12	7	18
30.000.000											19

To evaluate the effect of different parameters on the performance of the optimal strategy improvement algorithm, the experiments have been repeated with different outdegrees and numbers of colors. The figures in each field of the following table refer to random games with constant outdegree 3 / 12 / 40.

positions	10 colors	30 colors	100 colors	300 colors	1000 colors
1000	14 / 4 / 1	18 / 8 / 3	22 / 14 / 9	18 / 15 / 10	19 / 18 / 11
3000	19 / 3 / 1	25 / 16 / 2	30 / 28 / 10	31 / 28 / 16	32 / 25 / 14
10000	13 / 3 / 2	28 / 8 / 2	45 / 30 / 14	54 / 49 / 37	60 / 52 / 32
30000	14 / 4 / 1	32 / 9 / 2	81 / 27 / 6	110 / 83 / 62	125 / 106 / 63
100000	16 / 4 / 2	56 / 9 / 2	125 / 20 / 4	232 / 175 / 34	293 / 269 / 127
300000	19 / 4 / 2	50 / 9 / 2	102 / 21 / 4	401 / 333 / 12	648 / 536 / 290

For random graphs, the number of update games seems to depend mainly on the number of colors and the outdegree. The figures indicate that, for a given number of colors and a given outdegree, the number of iterations first increases with the number of positions, but, after a threshold that increases with the number of colors and decreases with increasing outdegree, decreases again, and swings in at a low value that increases with the number of colors and decreases with the outdegree. The complete figures, however, indicate that the variance (rather than the average number of iterations) decreases with an increasing number of positions.

Running Time. The longest running time for the optimal strategy improvement method in all benchmarks, 9 hours and 25 minutes, was observed on samples with 30 million positions, outdegree six, and 20 colors. However, the used implementation is a plain Java implementation that offers much leeway for optimizations.

3.4.2 Benchmarks

The low expected number of updates (which is confirmed by all following benchmarks) restricts the potential competitors: The randomized subexponential algorithms of Ludwig [Lud95], and Björklund and Vorobyov [BV07] perform exactly one update in every improvement step. It is therefore almost sure that the required number of update steps is at least linear in the size of the game. In case of Ludwig's algorithm, the update complexity is also much higher.

For the first benchmark, we restricted the focus on the algorithm of Vöge and Jurdziński [VJ00], and a (not subexponential) variant of the algorithm of

Björklund and Vorobyov [BV07], which, in every step, chooses a locally profitable modification uniformly at random for every position, for which a profitable modification exists.

The following table compares the expected number of iterations of our algorithm (*opt*) with the variant of Björklund and Vorobyov (*rand*) and Vöge and Jurdziński's algorithm (VJ) for random games with 3 colors and outdegree 6.

positions	30	100	300	1000	3000	10000	30000	100000	300000
opt	1.1	1.4	1.7	1.7	1.9	2.0	2.0	2.0	2.0
rand	2.5	2.9	3.1	3.0	3.0	3.1	3.2	3.7	4.0
VJ	5.3	12.2	26.1	66.1	182.0	573.1	1665.3[2]	—	—

The algorithm of Vöge and Jurdziński was not considered in the following benchmarks, because it took several days even for small random games with only 30000 positions and outdegree 6. This is partly due to the fact that the observed number of iterations grows linearly in the size of the game, and partly due to the much higher update complexity of $O(m\,n)$. (The predicted running time for every instance of the following benchmark is, for example, several decades.)

Different to the algorithm of Vöge and Jurdziński, the performance of the variant of Björklund and Vorobyov's algorithm (*rand*) on random games is close to the performance of the optimal improvement algorithm. The cost of the individual updates for *rand* is slightly higher, because 0 cycles need to be excluded in their approach, which results in higher numbers (by a factor linear in the size of the game). Together with the slightly smaller number of iterations, the running time of our algorithm improves over theirs by a factor of approximately 2 on the considered random games.

The following benchmarks investigates the effect of the number of colors, using random games with 300000 positions and branching degree 40. As in the previous benchmark, *opt* clearly improves over *rand*, but the factor remains small (2–3).

colors	3	5	10	17	30	55	100	175	300	550	1000
opt	1.0	1.0	1.4	2.0	2.0	4.0	2.9	23.7	6.7	9.9	24.6
rand	2.0	3.0	3.2	5.0	6.7	11.0	9.8	30.0	11.8	24.8	50.7

The difference between *opt* and *rand* becomes apparent once structure is added to the game. The following benchmark consists of random games with *index* colors, $index^2$ positions, and $index^3$ edges. The edges are not distributed uniformly; preference is given to close successors. The square of the observed

[2]For 30000 positions, each sample took approximately four days on a 2.6 GHz Dual Core AMD Opteron machine (as compared to 1.5 seconds for the optimal strategy improvement algorithm). The experiment was therefore terminated after ten samples.

number of updates of *rand* growth faster than the cube of the observed number of updates of *opt*.

index	10	20	30	40	50	60	70	80	90	100	110
opt	2.0	3.4	3.5	7.1	4.8	4.2	5.6	7.4	7.2	8.1	7.6
rand	4.2	8.5	11.0	17.5	23.4	21.9	25.0	29.1	27.9	30.6	38.5

The following table compares the performance of *opt* and *rand* on a benchmark that consists of random chains of sparsely linked subgames. While randomized, it provides much structure. On this benchmark, the number of updates needed by *rand* is quadratic in the number of updates needed by *opt*.

positions	74	299	1199	2699	4799	7499	10799	16874	27074
opt	3.3	6.5	11.2	26.3	25.4	35.8	34.8	59.8	63.5
rand	11.2	69.5	240.3	615.9	1354.4	1190.1	2118.2	3192.0	5876.4

The following table provides the figures for the analysis of a bipartite version of the $H_{index,index}$ games used in [Jur00] to estimate the worst case complexity of Jurdziński's algorithm. While this benchmark turns out not to be particularly hard for strategy improvement techniques, it still reveals some differences. (Constant vs. logarithmic number of iterations in the index or size of the game.)

index	2	4	8	16	32	64	128	256	512	1024	2048
opt	2	2	2	2	2	2	2	2	2	2	2
rand	2.7	3.9	5.3	6.9	8.6	10.3	12.1	14.0	15.8	17.5	19.4

For (Co-)Büchi games, the difference between the algorithms is particularly interesting, because *opt* coincides with standard fixed point algorithms [McN93, EL86, Zie98] in this case. For this reason, a dedicated benchmark for (Co-)Büchi games is included. On this benchmark, the number of updates needed by *rand* is approximately quadratic in the number of updates needed by *opt*.

positions	506	2027	4562	7499	10799	14699	19199	24299
opt	13	26	39	50	60	70	80	90
rand	170.8	677.6	1522.5	2501.6	3601.3	4901.6	6401.5	8101.5

The last benchmark for parity games tests the sensitivity of strategy improvement algorithms to 'traps' that lure them to a wrong direction. While not representative, it is a nice demonstration for the difference that choosing the best of 2^n combinations may make compared to choosing a random combination.

positions	11	35	101	335	1001	3335	10001	33335
opt	1	1	1	1	1	1	1	1
rand	5.1	13.7	35.7	113.4	335.4	1113.6	3335.6	11113.5

Finally, we turn back to random games, and compare the performance of both algorithms for the analysis of mean payoff games with outdegree six and small payoffs. (The extension of the game-based approach to mean payoff games is trivial.) The number of iterations needed by *rand* for finding the 0 mean partition is more than cubic in the number of iterations needed by *opt*.

positions	30	100	300	1000	3000	10000	30000	100000	300000
opt	1.2	1.5	1.7	2.1	2.3	2.8	3.5	3.2	3.4
rand	1.6	3.4	5.0	8.8	19.6	36.8	43.0	44.9	57.0

The benchmarks show that, with or without structure, the expected number of iterations needed by the optimal improvement algorithm is very small. For simple instances, the advantage of *opt* over *rand* is limited to a small constant. However, in all benchmarks with structure the average number of iterations needed by *rand* growth much faster than the average number of iterations needed by *opt*, for example, quadratic in the fourth and sixth benchmark, and cubic in the last.

3.5 Discussion

The applicability of strategy improvement algorithms crucially depends on the quality of the individual improvement step, or, likewise, on the expected amount of improvement steps. Current strategy improvement techniques suffer from deficiencies in estimating the effect of strategy modifications: They cannot predict their global effect, let alone the cross effect between different modifications. The suggested game-based optimal strategy improvement algorithm overcomes this deficiency, and allows us, for the first time, to make full use of the advantages attached to concurrent strategy modifications.

Besides the better results on different benchmarks, the game-based approach provides additional advantages over current strategy improvement algorithms. While current strategy improvement algorithms behave unintuitively on Büchi and Co-Büchi games (which may well undermine the trust in their performance), the game-based approach coincides with traditional fixed point algorithms on this class of parity games. It can therefore be viewed as an alternative generalization from single to nested fixed points. A further advantage is the strong improvement guarantee provided by the game-based approach. The estimation v_i that results from the *i-th* improvement step is better than all estimations that refer to any restriction of the moves of player 0 to some previous improvement arena $\mathcal{A}_{v_j}$ $(j < i)$. This is in contrast to current strategy improvement techniques, which only guarantee to construct a sequence of improving estimations.

From a practical point of view, the amount of improvement steps used by simplex style algorithms tends to be linear in the amount of constraints that define the simplex [Sma83]. In our algorithm, the edges that originate from positions of player 0 define these constraints. In several benchmarks, approximately 30% (at the end) to 50% (at the beginning) of the edges that originate from positions of player 0 are improvement edges, and the expected number of improvements in every step is *linear* in the size of the improvement graph, which leads, for example, to the constant number of updates in random games. Interestingly, the discussed optimal improvement algorithm works particularly well in situations that are most critical for traditional simplex style algorithms: Situations where a wide variety of choices perturb the pivot rule, in particular in the case where only stale modifications exist (being on a large plateau with equivalently valuated corners). Taking into account that elliptic methods [Kha79] have never been earnestly considered to replace the simplex algorithm, and even Karmarkar's interior point method [Kar84] could not supersede the simplex method, polynomial time algorithms would need to satisfy high demands to become practically important even if the optimal strategy improvement should turn out to be exponential.

While the update complexity of the algorithms is low ($O(m + n \log n)$ arithmetic operations), finding a non-trivial bound for the overall complexity, that is, finding a bound on the number of updates, remains an intriguing future challenge. Intuitively, it seems that the worst case scenario for the optimal improvement algorithm occurs when the algorithm can never play to its strengths, that is, if there is never a variety of improvements to select from. Interestingly, it is this seemingly worst case scenario, for which a non-trivial upper bound can easily be provided: For every optimal strategy of player 0, at least one edge is a new improvement edge. No variety therefore implies that this improvement edge is selected, while all previously selected edges of the strategy remain. Thus, the algorithm needs less than n strategy updates in this scenario. This $O(n)$ bound for the number of updates is currently the best estimation from below for the improvement complexity.

This leads to the intriguing open complexity problem of infinite games over finite structures: Solving parity games (and thus the μ-calculus model checking problem [Wil01]), as well as the analysis of more general infinite games over finite structures like mean payoff games, are among the few problems that are known to be in $\mathbf{NP} \cap \mathbf{coNP}$ [McN93] (and even in $\mathbf{UP} \cap \mathbf{coUP}$ [ZP96, Jur98]), but whose membership in $\mathbf{P}$ is still an open problem. While the membership of randomized strategy improvement algorithms [Lud95, BV07] to the class $\mathbf{ZPP}$ is open (while they are known not to be in $\mathbf{P}$), the proposed algorithm

may turn out to be polynomial. Understanding the complexity of the game-based strategy improvement algorithm would either lead to a proof that parity / mean payoff games can be solved in polynomial time, or would greatly help to understand the hardness of these problems. Hardness proofs for game-base strategy improvement, however, will not be simple. It took a quarter of a century to find a family of examples, for which the complexity of the simplex algorithm is exponential [KM72]. These classical examples from linear programming do not extend to game-based improvement methods; the Klee Minty polytope [KM72], for example, requires only a single update step from the origin (and at most linearly many steps from any arbitrary corner of the polytope) if we can consider all combinations of profitable and stale base changes in every improvement step.

Part II

Logics & Automata

Overview

A main challenge in system design is the construction of correct implementations from their temporal specifications. Traditionally, system design consists of three separated phases, the specification phase, the implementation phase, and the validation phase. From a scientific point of view it seems inviting to overcome the separation between the implementation and validation phase, and to replace the manual implementation of a system and its subsequent validation by a push-button approach, which automatically synthesizes an implementation that is correct by construction. Automating the system construction also provides valuable additional information: We can distinguish *unsatisfiable* system specifications, which otherwise would go unnoticed, leading to a waste of effort in the fruitless attempt of finding a correct implementation.

One important choice on the way towards the ideal of fully automated system construction is the choice of the specification language. For temporal specifications, three different types of logics have been considered: Linear-time logics [Wol82], branching-time logics [CE82], and alternating-time logics [AHK02]. The different paradigms are suited for different types of systems and different design phases. Linear-time logic can only reason about properties of all possible runs of a system. Consequently, it cannot express the existence of different runs. A constructive non-emptiness test of an LTL specification is therefore bound to create a system that has exactly one possible run. Branching-time logics [CE82], on the other hand, can reason about *possible* futures, but they refer to *closed systems* [AHK02] that do not distinguish between different participating agents. Finally, alternating-time logics [AHK02] can reason about the strategic capabilities of groups of agents to cooperate to obtain a temporal goal.

As an example that illustrates the differences, let us consider a vending machine that offers coffee and tea. Ultimately, we want the machine to react on the requests of a customer, who shall be provided with coffee or tea upon request. In the alternating-time temporal logic ATL*, we have a natural correspondence to

this requirement: We simply specify that the customer has the strategic capability to get coffee or tea from the vending machine, without the need of cooperation, written $\langle\langle \text{customer} \rangle\rangle \bigcirc \text{get}_{coffee}$ or $\langle\langle \text{customer} \rangle\rangle \bigcirc \text{get}_{tea}$, respectively.

In branching-time logic, there is no natural correspondence to the property. The typical approximation is to specify the *possibility* that coffee or tea is provided, written $E \bigcirc \text{get}_{coffee}$ or $E \bigcirc \text{get}_{tea}$, respectively. However, this does no longer guarantee that the customer can choose; the specification is also fulfilled if her health insurance company can override her decision for coffee. A workaround may be to introduce a dedicated communication interface between the vending machine and the customer, and represent the desire for coffee by a desire_{coffee} bit controlled by the customer. The property may then be approximated by $E \bigcirc \text{desire}_{coffee}$ and $\text{desire}_{coffee} \rightarrow A \bigcirc \text{get}_{coffee}$. In LTL, the possibility of different system behaviors cannot be expressed within the logic. Here, in addition to specifying an interface, we would have to distinguish between parts of the system under our control (the vending machine) and parts outside of our control (the customer). A possible approximation would be $\text{desire}_{coffee} \rightarrow \bigcirc \text{get}_{coffee}$, with the addition that there is not only the need to design an interface to the customer beforehand, but also to make assumptions about her behavior.

The usage of linear or branching-time logics thus requires to first establish the static interfaces between the system components. This restriction allows for the application of synthesis techniques only after the central design problem of establishing the interfaces between the individual components has been completed, instead of starting in an earlier specification phase with a more abstract view on the system. If synthesis fails, we cannot distinguish unrealizable specifications from interface design errors.

Alternating-time logics, on the other hand, are exceptionally well suited for the specification of distributed systems on such an abstract level. At this level of the design process, checking the satisfiability of a formula in alternating-time logic means testing the consistency of the system specification. Satisfiability checking can, for example, be used to demonstrate that it is impossible to implement fair contract-signing without a trusted third party [EY80].

This part of the thesis provides an automata-theoretic decision procedures and tight complexity bounds for the realizability problem of ATμC (Chapter 4) and its semantic sublogic ATL* (Chapter 5). We show that the satisfiability problem for ATμC and ATL* is not harder than the satisfiability problem for the classic μ-calculus and CTL*, respectively. This low complexity is particularly interesting for ATL*: Here, the satisfiability problem and its constructive extension to model construction is no more expensive than model checking [AHK02, dAHM01] – both problems are 2EXPTIME-complete.

Chapter 4

Satisfiability and Finite Model Property of the Alternating-Time μ-Calculus

Abstract

In this chapter, a decision procedure for the alternating-time μ-calculus is proposed. The decision procedure is based on a representation of alternating-time formulas as a novel type of alternating automata, automata over concurrent game structures. We show that language emptiness of these automata can be checked in exponential time. The complexity of our construction meets the known lower bounds for deciding the satisfiability of the classic μ-calculus. It follows that the satisfiability problem is EXPTIME-complete for the alternating-time μ-calculus.

4.1 Introduction

In the design of distributed protocols, we are often interested in the strategic abilities of certain agents. For example, in a contract-signing protocol, it is important to ensure that, while Alice and Bob can cooperate to sign a contract, Bob never has a strategy to obtain Alice's signature unless, at the same time, Alice has a strategy to obtain Bob's signature as well (cf. [KR03]). Such proper-

ties can be expressed in the alternating-time μ-calculus (ATμC) [AHK02], which extends the classic μ-calculus with modalities that quantify over the strategic choices of a group of agents. The models of ATμC are a special type of labeled transition systems, called *concurrent game structures*, where each transition results from a set of decisions, one for each agent.

In this chapter, the first decision procedure for the satisfiability of ATμC formulas is presented. The *satisfiability* problem asks for a given ATμC formula φ whether there exists a concurrent game structure that satisfies φ. Previous research has focused on the model checking problem [AHK02, AHM$^+$98], which asks whether a *given* concurrent game structure satisfies its specification. By contrast, the suggested algorithm checks whether a specification can be implemented at all. It can, for example, be used to automatically prove the classic result that it is *impossible* to implement fair contract-signing without a trusted third party [EY80].

We introduce an automata-theoretic framework for alternating-time logics. *Automata over concurrent game structures* (ACGs) are a variant of alternating tree automata, where the atoms in the transition function do not refer to individual successors in the input structure, but instead quantify universally or existentially over all successors that result from the agents' decisions. Specifically, a *universal* atom $(\Box, A')$ refers to *all* successor states for *some* decision of the agents in a set A', and an *existential* atom $(\Diamond, A')$ refers to *some* successor state for *each* decision of the agents *not* in A'. In this way, the automaton can run on game structures with arbitrary, even infinite, branching degree. Every ATμC formula can be translated into an automaton that accepts exactly the models of the formula. Satisfiability of ATμC formulas thus corresponds to language non-emptiness of ACGs.

The core result of this chapter is the finite model property for ACGs. We first prove that, given any game structure accepted by an ACG $\mathcal{G}$, we can find a *bounded* game structure that is also accepted by $\mathcal{G}$. In the bounded game structure, the number of possible decisions of each agent is limited by some constant m, determined by the size of $\mathcal{G}$.

The emptiness problem of ACGs thus reduces to the emptiness problem of standard alternating tree automata and, since non-empty automata over finitely-branching structures always accept some finite structure [Rab72, MS95], there must exist a finite game structure in the language of a non-empty ACG $\mathcal{G}$. The drawback of this reduction is that the trees accepted by the alternating tree automaton branch over the decisions of *all* agents: The number of directions is therefore exponential in the number of agents. Since the emptiness problem of

alternating tree automata is exponential in the number of directions, this results in a double-exponential decision procedure.

We show that it is possible to decide emptiness in single-exponential time. Instead of constructing an *alternating* automaton that accepts exactly the m-bounded game structures in the language of the ACG, we construct a *universal* automaton that only preserves emptiness. Unlike alternating automata, universal automata can be reduced to deterministic automata with just a single exponential increase in the number of states. For deterministic automata, the size of the emptiness game is only linear in the number of directions.

Our approach is constructive and yields a tight complexity bound: The satisfiability problem for ATμC is EXPTIME-complete. If the ATμC formula is satisfiable, we can synthesize a finite model within the same complexity bound.

Since ATμC subsumes the alternating-time temporal logic ATL* [dAHM01, AHK02], we obtain a decision procedure for this logic as well.

Related work. The automata-theoretic approach to the satisfiability problem was initiated in the classic work by Büchi, McNaughton, and Rabin on monadic second-order logic [Büc62, McN66, Rab72]. For linear-time temporal logic, satisfiability can be decided by a translation to automata over infinite words [VW94]; for branching-time logics, such as CTL* and the modal μ-calculus, by a translation to automata over infinite trees that branch according to inputs and nondeterministic choices [EJ91, KVW00, KV00, Wil01]. For alternating-time temporal logics, previous decidability results have been restricted to ATL [vD03, WLWW06], a sublogic of ATL*.

Automata over concurrent game structures, introduced in this thesis, provide an automata-theoretic framework for alternating-time logics. Automata over concurrent game structures extend symmetric alternating automata [Wil01], which have been proposed as the automata-theoretic framework for the classic μ-calculus. Symmetric automata branch universally into all successors or existentially into some successor.

4.2 Preliminaries

4.2.1 Concurrent Game Structures

Concurrent game structures [AHK02] generalize labeled transition systems to a setting with multiple agents. A *concurrent game structure* (CGS) is a tuple $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$, where

- Π is a finite set of atomic propositions,
- A is a finite set of agents,
- S is a set of states, with a designated initial state $s_0 \in S$,
- $l : S \to 2^{\Pi}$ is a labeling function that decorates each state with a subset of the atomic propositions,
- Δ is a set of possible decisions for every agent[1], and
- $\tau : S \times \Delta^A \to S$ is a transition function that maps a state and the decisions of the agents to a new state.

A concurrent game structure is called *bounded* if the set Δ of decisions is finite, *m-bounded* if $\Delta = \mathbb{N}_m = \{1, \ldots, m\}$, and *finite* if S and Δ are finite.

Example. As a running example, we introduce a simple CGS $\mathcal{C}_0$ with an uncountable number of states and an uncountable number of possible decisions (cf. Figure 4.1). In every step, two agents each pick a real number and move to the state ${d_2}^2 - {d_1}^2$, where d_1 is the decision of agent a_1 and d_2 is the decision of agent a_2. We use two propositions, p_1 and p_2, where p_1 identifies the non-negative numbers and p_2 the rational numbers. Let $\mathcal{C}_0 = (\Pi, A, S, s_0, l, \Delta, \tau)$ denote the CGS with $\Pi = \{p_1, p_2\}$, $A = \{a_1, a_2\}$, $S = \mathbb{R}$, $s_0 = 0$, $p_1 \in l(s)$ if and only if $s \geq 0$, $p_2 \in l(s)$ if and only if $s \in \mathbb{Q}$, $\Delta = \mathbb{R}$, and $\tau : (s, (d_1, d_2)) \mapsto {d_2}^2 - {d_1}^2$. It is easy to see that, in all states of this CGS, agent a_1 can enforce that p_1 eventually always holds true. Additionally, if agent a_1 decides before agent a_2, agent a_2 can always respond with a decision such that p_2 holds in the following state.

4.2.2 Alternating-Time μ-calculus

The *alternating-time μ-calculus* (ATμC) extends the classic μ-calculus with modal operators which express that an agent or a coalition of agents has a strategy to accomplish a goal. ATμC formulas are interpreted over concurrent game structures.

[1]The restriction that the set of decisions is independent of the position and the agent is only applied to simplify the argumentation in this part. The computation trees considered in the following part are concurrent game structures where the set of decisions depend on both, the agent and the position.

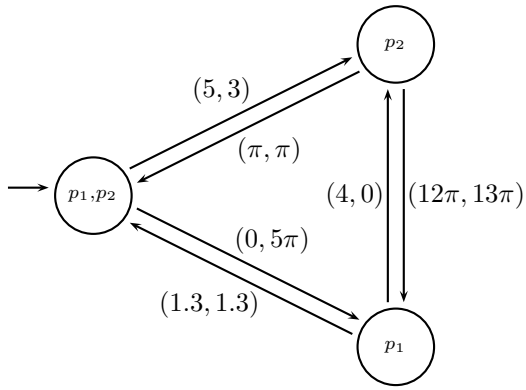


Figure 4.1: The running example is the concurrent game structure $\mathcal{C}_0$ with two agents, whose states and decisions consist of the real numbers $\mathbb{R}$, with initial state $s_0 = 0$. In every step, the two agents a_1 and a_2 pick a real number (d_1 and d_2, respectively) and move to the state ${d_2}^2 - {d_1}^2$. (That is, the transition function does not depend on the current state.) $\mathcal{C}_0$ has two propositions ($\Pi = \{p_1, p_2\}$) that mark the non-negative (p_1) and rational (p_2) numbers, respectively. This figure shows a fragment of $\mathcal{C}_0$, which includes the initial state and the states 25π and -16, and a fragment of the transitions between these states.

ATμC Syntax. ATμC contains the modality $\Box_{A'}\varphi$, expressing that a set $A' \subseteq A$ of agents can enforce that a property φ holds in the successor state, and the modality $\Diamond_{A'}\varphi$, expressing that it cannot be enforced against the agents A' that φ is violated in the successor state. Let Π and B denote disjoint finite sets of atomic propositions and bound variables, respectively. Then

- *true* and *false* are ATμC formulas.
- p, $\neg p$ and x are ATμC formulas for all $p \in \Pi$ and $x \in B$.
- If φ and ψ are ATμC formulas then $\varphi \wedge \psi$ and $\varphi \vee \psi$ are ATμC formulas.
- If φ is an ATμC formula and $A' \subseteq A$ then $\Box_{A'}\varphi$ and $\Diamond_{A'}\varphi$ are ATμC formulas.
- If $x \in B$ and φ is an ATμC formula where x occurs only free, then $\mu x.\varphi$ and $\nu x.\varphi$ are ATμC formulas.

The set of subformulas of a formula φ is denoted by $sub(\varphi)$ and its alternation depth [Bra96] by $alt(\varphi)$. For simplicity, we use the syntactic alternation depth for our constructions.

ATμC Semantics. An ATμC formula φ with atomic propositions Π is interpreted over a CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$. $\|\varphi\|_{\mathcal{C}} \subseteq S$ denotes the set of states where φ holds. A CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is a *model* of a specification φ with atomic propositions Π if and only if $s_0 \in \|\varphi\|_{\mathcal{C}}$.

- Atomic propositions are interpreted as follows: $\|\mathit{false}\|_{\mathcal{C}} = \emptyset$ and $\|\mathit{true}\|_{\mathcal{C}} = S$, $\|p\|_{\mathcal{C}} = \{s \in S \mid p \in l(s)\}$ and $\|\neg p\|_{\mathcal{C}} = \{s \in S \mid p \notin l(s)\}$.
- Conjunction and disjunction are interpreted as intersection and union, respectively: $\|\varphi \wedge \psi\|_{\mathcal{C}} = \|\varphi\|_{\mathcal{C}} \cap \|\psi\|_{\mathcal{C}}$ and $\|\varphi \vee \psi\|_{\mathcal{C}} = \|\varphi\|_{\mathcal{C}} \cup \|\psi\|_{\mathcal{C}}$.
- A state $s \in S$ is in $\|\Box_{A'}\varphi\|_{\mathcal{C}}$ if and only if the agents A' can make a decision $\upsilon \in \Delta^{A'}$ such that, for all decisions $\upsilon' \in \Delta^{A \setminus A'}$, φ holds in the successor state:
  $\|\Box_{A'}\varphi\|_{\mathcal{C}} = \{s \in S \mid \exists \upsilon \in \Delta^{A'}.\ \forall \upsilon' \in \Delta^{A \setminus A'}.\ \tau(s, (\upsilon, \upsilon')) \in \|\varphi\|_{\mathcal{C}}\}$.
- A state $s \in S$ is in $\|\Diamond_{A'}\varphi\|_{\mathcal{C}}$ if and only if for all decisions $\upsilon \in \Delta^{A \setminus A'}$ of the agents not in A', the agents in A' have a counter decision $\upsilon' \in \Delta^{A'}$ which ensures that φ holds in the successor state:
  $\|\Diamond_{A'}\varphi\|_{\mathcal{C}} = \{s \in S \mid \forall \upsilon' \in \Delta^{A \setminus A'}.\ \exists \upsilon \in \Delta^{A'}.\ \tau(s, (\upsilon, \upsilon')) \in \|\varphi\|_{\mathcal{C}}\}$.
- The least and greatest fixed points are interpreted as follows:
  - $\|\mu x.\varphi\|_{\mathcal{C}} = \bigcap\{S_x \subseteq S \mid \|\varphi\|_{\mathcal{C}_x^{S_x}} \subseteq S_x\}$, and
  - $\|\nu x.\varphi\|_{\mathcal{C}} = \bigcup\{S_x \subseteq S \mid \|\varphi\|_{\mathcal{C}_x^{S_x}} \supseteq S_x\}$,

  where $\mathcal{C}_x^{S_x} = (\Pi \cup \{x\}, A, S, s_0, l_x^{S_x}, \Delta, \tau)$ denotes the modified CGS with the labeling function $l_x^{S_x} : S \to 2^{\Pi \cup \{x\}}$ with $l_x^{S_x}(s) \cap \Pi = l(s)$ and $x \in l_x^{S_x}(s) \Leftrightarrow s \in S_x \subseteq S$. Since the bound variable x occurs only positive in φ, $\|\varphi\|_{\mathcal{C}_x^{S_x}}$ is monotone in S_x and the fixed points are well-defined.

ATμC contains the classic μ-calculus with the modal operators $\Box$ and $\Diamond$, which abbreviate $\Box_\emptyset$ and $\Diamond_A$, respectively. ATμC also subsumes the temporal logic ATL* [AHK02], which is the alternating-time extension of the branching-time temporal logic CTL*. ATL* contains the path quantifier $\langle\langle A' \rangle\rangle$, which ranges over all paths the players in A' can enforce. There is a canonical translation from ATL* to ATμC [dAHM01].

Example. As discussed in Section 4.2.1, the example CGS $\mathcal{C}_0$ has the property that, in all states, agent a_1 can enforce that p_1 eventually always holds true, and agent a_2 can respond to any decision of agent a_1 with a counter decision such that p_2 holds in the following state. This property is expressed by the ATμC formula $\psi = \nu x.(\mu y.\nu z.\Box_{\{a_1\}}(p_1 \wedge z \vee y)) \wedge \Diamond_{\{a_2\}} p_2 \wedge \Diamond_{\emptyset} x$.

4.2.3 Automata over Finitely Branching Structures

An *alternating parity automaton* with a finite set Υ of directions is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is a designated initial state, δ is a transition function, and $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive Boolean combination of states and directions.

In particular, we consider alternating parity automata that run on bounded CGSs with a fixed set Π of atomic propositions ($\Sigma = 2^{\Pi}$), a fixed set A of agents and a fixed finite set Δ of decisions ($\Upsilon = \Delta^A$). The acceptance mechanism is defined in terms of run trees. As usual, an Υ-tree is a prefix-closed subset $Y \subseteq \Upsilon^*$ of the finite words over the set Υ of directions. For given sets Σ and Υ, a Σ-*labeled* Υ-*tree* is a pair $\mathcal{C}$, consisting of a tree $Y \subseteq \Upsilon^*$ and a labeling function $l : Y \to \Sigma$ that maps every node of Y to a letter of Σ. If Υ and Σ are not important or clear from the context, $\mathcal{C}$ is called a tree.

A *run tree* $\langle R, r\rangle$ on a given CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is a $Q \times S$-labeled tree whose root is decorated with $r(\varepsilon) = (q_0, s_0)$, and for each node $n \in R$ that is decorated with a label $r(n) = (q, s)$, there is a set $\mathfrak{A}_n \subseteq Q \times \Upsilon$ that satisfies $\delta(q, l(s))$, such that (q', υ) is in $\mathfrak{A}_n$ if and only if some child of n is decorated with a label $(q', \tau(s, \upsilon))$.

A run tree is *accepting* if and only if all infinite paths fulfill the *parity condition*. An infinite path fulfills the parity condition if and only if the highest color of the states appearing infinitely often on the path is even. A CGS is *accepted* by the automaton if and only if it has an accepting run tree. The set of CGSs accepted by an automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is called empty if and only if its language is empty.

The acceptance of a given CGS $\mathcal{C}$ can also be viewed as the outcome of a *game* played over $Q \times S$, starting in (q_0, s_0). When the game reaches a position (q, s), player *accept* first chooses a set $\mathfrak{A} \subseteq Q \times \Upsilon$ of atoms that satisfies $\delta(q, l(s))$. Player *reject* then chooses one atom (q', υ) from $\mathfrak{A}$ and the game continues in $(q', \tau(s, \upsilon))$. An infinite sequence $(q_0, s_0)(q_1, s_1)(q_2, s_2)\ldots$ of game positions is called a *play*. A play is *winning* for player *accept* if and only if it satisfies

the parity condition. A *strategy* for player *accept* (*reject*) maps each history of decisions of both players to a decision of player *accept* (*reject*). A pair of strategies determines a play. A strategy for player *accept* is *winning* if and only if, for all strategies of player *reject*, the play determined by the strategies is winning for player *accept*. The CGS $\mathcal{C}$ is accepted if and only if player *accept* has a winning strategy.

An alternating automaton is called

- *universal* if the image of δ consists only of conjunctions,
- *nondeterministic* if the image of δ consists only of formulas that, when rewritten into disjunctive minimal form, contain, in each disjunct, at most one element of $Q \times \{\upsilon\}$ for each $\upsilon \in \Upsilon$, and
- *deterministic* if it is universal and nondeterministic.

For nondeterministic automata, emptiness can be checked with an *emptiness game* over Q where, instead of considering the letter $l(s)$ on some state s of a given CGS, the letter is chosen by player *accept*. The nondeterministic automaton is non-empty if and only if player *accept* has a winning strategy in the emptiness game.

An alternating automaton is called

- a *Büchi* automaton if the image of $\alpha(Q) \subseteq \{1, 2\}$ is contained in $\{1, 2\}$,
- a *Co-Büchi* automaton if the image of $\alpha(Q) \subseteq \{0, 1\}$ is contained in $\{0, 1\}$,
- a *safety* automaton if the image of $\alpha(Q) = \{0\}$ is $\{0\}$, or, likewise, if all run-trees are accepting (for safety automata, α is therefore omitted), and
- a *weak* automaton, if, for every path on every run tree, the color increases monotonously, that is, if $\forall(q, \sigma) \in Q \times \Sigma \forall(q', \upsilon) \in \delta(q, \sigma).\, \alpha(q') \geq \alpha(q)$ holds true.

For Büchi and Co-Büchi automata, α is often represented indirectly by the set F of states with higher color ($F = \alpha^{-1}(2)$ for Büchi, and $F = \alpha^{-1}(1)$ for Co-Büchi automata).

If Υ is singleton, it is omitted in the notation, and $\mathcal{A} = (\Sigma, Q, q_0, \delta : Q \times \Sigma \to \mathbb{B}^+(Q), \alpha)$ is called a *word* automaton.

4.3 Automata over Concurrent Game Structures

In this section, we introduce *automata over concurrent game structures* (ACGs) as an automata-theoretic framework for the alternating-time μ-calculus. The automata over finitely branching structures described in Subsection 4.2.3 do not suffice for this purpose, because they are limited to bounded CGSs. Generalizing symmetric automata [Wil01], ACGs contain *universal atoms* $(\Box, A')$, which refer to *all* successor states for *some* decision of the agents in A', and *existential atoms* $(\Diamond, A')$, which refer to *some* successor state for *each* decision of the agents *not* in A'. In this way, ACGs can run on CGSs with an arbitrary, even infinite, number of decisions.

An ACG is a tuple $\mathcal{G} = (\Sigma, Q, q_0, \delta, \alpha)$, where Σ, Q, q_0, and α are defined as for alternating parity automata in the previous section. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times (\{\Box, \Diamond\} \times 2^A) \cup \{\varepsilon\}))$ now maps a state and an input letter to a positive Boolean combination of three types of atoms: $(\Box, A')$ is a universal atom, $(\Diamond, A')$ is an existential atom, and ε is an ε-transition, where only the state of the automaton is changed and the state of the CGS remains unchanged.

An ACG is called *ε-free* if it has no ε-transitions, and *universal*, if the mapping of δ is contained in $\mathbb{B}^+(Q \times \{(\Box, A)\} \cup \{\varepsilon\})$[2].

A run tree $\langle R, r\rangle$ on a given CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is a $Q \times S$-labeled tree where the root is labeled with (q_0, s_0) and where, for a node n with a label (q, s) and a set $L = \{r(n \cdot \rho) \mid n \cdot \rho \in R\}$ of labels of its successors, the following property holds: There is a set $\mathfrak{A} \subseteq Q \times (\{\Box, \Diamond\} \times 2^A \cup \{\varepsilon\})$ of atoms satisfying $\delta(q, l(s))$ such that

- for all universal atoms $(q', \Box, A')$ in $\mathfrak{A}$, there exists a decision $\upsilon \in \Delta^{A'}$ of the agents in A' such that, for all counter decisions $\upsilon' \in \Delta^{A \setminus A'}$, $(q', \tau(s, (\upsilon, \upsilon'))) \in L$,
- for all existential atoms $(q', \Diamond, A')$ in $\mathfrak{A}$ and all decisions $\upsilon' \in \Delta^{A \setminus A'}$ of the agents not in A', there exists a counter decision $\upsilon \in \Delta^{A'}$ such that $(q', \tau(s, (\upsilon, \upsilon'))) \in L$, and
- for all ε-transitions (q', ε) in $\mathfrak{A}$, $(q', s) \in L$.

[2] Note that universal ACGs do *not* correspond to universal alternating automata. While the latter can only recognize trace languages, universal ACGs correspond to universal specification languages.

As before, a run tree is accepting if and only if all paths satisfy the parity condition, and a CGS is accepted if and only if there exists an accepting run tree.

The acceptance of a CGS can again equivalently be defined as the outcome of a game over $Q \times S$, starting in (q_0, s_0). Each round of the game now consists of two stages. In the first stage, player *accept* chooses a set $\mathfrak{A}$ of atoms satisfying $\delta(q, l(s))$, and player *reject* picks one atom from $\mathfrak{A}$. If the result of the first stage is an ε-transition (q', ε), then the round is finished and the game continues in (q', s) with the new state of the automaton. If the result of the first stage is a universal atom $(q', (\Box, A'))$, the second stage begins by player *accept* making the decisions $\upsilon \in \Delta^{A'}$ for the agents in A', followed by player *reject* making the decisions $\upsilon' \in \Delta^{A \setminus A'}$ for the remaining agents. Finally, if the result of the first stage is an existential atom $(q, (\Diamond, A'))$, the order of the two choices is reversed: First, player *reject* makes the decisions $\upsilon' \in \Delta^{A \setminus A'}$ for the agents in $A \setminus A'$; then, player *accept* makes the decisions $\upsilon \in \Delta^{A'}$ for the players in A'. After the decisions are made, the game continues in $(q', \tau(s, (\upsilon, \upsilon')))$.

A winning strategy for player *accept* uniquely defines an accepting run tree, and the existence of an accepting run tree implies the existence of a winning strategy. The game-theoretic characterization of acceptance is often more convenient than the characterization through run trees, because parity games are memoryless determined [EJ91]. A CGS is therefore accepted by an ACG if and only if player *accept* has a memoryless winning strategy in the acceptance game, that is, if and only if she has a strategy where her choices only depend on the state of the game and the previous decisions in the current round.

As an additional complexity measure for an ACG $\mathcal{G}$, we use the set $atom(\mathcal{G}) \subseteq Q \times (\{\Box, \Diamond\} \times 2^A \cup \{\varepsilon\})$ of atoms that actually occur in some Boolean function $\delta(q, \sigma)$. The elements of $atom(\mathcal{G})$ are called the *atoms of* $\mathcal{G}$.

Example. The CGSs that satisfy the ATμC formula $\psi = \nu x.(\mu y.\nu z.\, \Box_{\{a_1\}}(p_1 \wedge z \vee y)) \wedge \Diamond_{\{a_2\}} p_2 \wedge \Diamond_{\emptyset} x$ from Section 4.2.2 are recognized by the ACG $\mathcal{G}_\psi = (\Sigma, Q, q_0, \delta, \alpha)$, where $\Sigma = 2^{\{p_1, p_2\}}$ and $Q = \{q_0, q_\mu, q_\nu, q_{p_2}\}$. The transition function δ maps

- (q_{p_2}, σ) to *true* if $p_2 \in \sigma$, and to *false* otherwise,
- (q_μ, σ) and (q_ν, σ) to $(q_\nu, \Box, \{a_1\})$ if $p_1 \in \sigma$, and to $(q_\mu, \Box, \{a_1\})$ otherwise, and
- (q_0, σ) to $\delta(q_\mu, \sigma) \wedge (q_{p_2}, \Diamond, \{a_2\}) \wedge (q_0, \Diamond, \emptyset)$.

The coloring function α maps q_μ to 1 and the remaining states to 0.

Consider again the example CGS $\mathcal{C}_0$ from Section 4.2.1, which satisfies ψ. In the acceptance game of $\mathcal{G}_\psi$ for $\mathcal{C}_0$, player *accept* has no choice during the first stage of each move, and can win the game by making the following decisions during the second stage:

- If one of the atoms $(q_\mu, \Box, \{a_1\})$ or $(q_\nu, \Box, \{a_1\})$ is the outcome of the first stage, agent a_1 makes the decision 0.
- If the atom $(q_{p_2}, \Diamond, \{a_2\})$ is the outcome of the first stage and agent a_1 has made the decision d_1, agent a_2 chooses $d_2 = d_1$.
- For all other atoms $(q, \circ, A')$, the decision for all agents in A' is 0.

4.3.1 From ATμC Formulas to Automata over Concurrent Game Structures

The following construction provides a translation of ATμC formulas to equivalent ACGs. It generalizes the construction for the modal μ-calculus suggested in [Wil01] and can be proved analogously.

Theorem 4.1 *Given an ATμC formula φ, we can construct an ACG $\mathcal{G}^\varepsilon_\varphi = (2^\Pi, sub(\varphi), \varphi, \delta, \alpha)$ with $|sub(\varphi)|$ states and atoms and $O(|alt(\varphi)|)$ colors that accepts exactly the models of φ.*

Construction: Without loss of generality, we assume that the bound variables have been consistently renamed to ensure that for each pair of different subformulas $\lambda x.\psi$ and $\lambda' x'.\psi'$ $(\lambda, \lambda' \in \{\mu, \nu\})$ of φ, the bound variables are different $(x \neq x')$.

- The transition function δ is defined, for all free variables p and all bound variables x, by
  - $\delta(p, \sigma) = \mathit{true}$, $\delta(\neg p, \sigma) = \mathit{false} \quad \forall p \in \sigma$;
  - $\delta(\neg p, \sigma) = \mathit{true}$, $\delta(p, \sigma) = \mathit{false} \quad \forall p \in \Pi \setminus \sigma$;
  - $\delta(\varphi \wedge \psi, \sigma) = (\varphi, \varepsilon) \wedge (\psi, \varepsilon)$ and $\delta(\varphi \vee \psi, \sigma) = (\varphi, \varepsilon) \vee (\psi, \varepsilon)$;
  - $\delta(\Box_{A'} \varphi, \sigma) = (\varphi, (\Box, A'))$ and $\delta(\Diamond_{A'} \varphi, \sigma) = (\varphi, (\Diamond, A'))$;
  - $\delta(x, \sigma) = (\lambda x.\varphi, \varepsilon)$ and $\delta(\lambda x.\varphi, \sigma) = (\varphi, \varepsilon) \quad \lambda \in \{\mu, \nu\}$.
- The coloring function α maps every subformula that is not a fixed point formula to 0. The colors of the fixed point formulas are defined inductively:

- Every least fixed point formula $\mu p.\psi$ is colored by the smallest odd color that is greater or equal to the highest color of each subformula of ψ.
- Every greatest fixed point formula $\nu p.\psi$ is colored by the smallest even color that is greater or equal to the highest color of each subformula of ψ. □

4.3.2 Eliminating ε-Transitions

Given an ACG $\mathcal{G}^\varepsilon = (\Sigma, Q, q_0, \delta, \alpha)$ with ε-transitions, we can find an ε-free ACG that accepts the same language. The idea of our construction is to consider the sequences of transitions from some position of the acceptance game that *exclusively* consist of ε-transitions: If the sequence is infinite, we can declare the winner of the game without considering the rest of the game; if the sequence is finite, we skip forward to the next non-ε-atom.

For a state $s \in S$ of an CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$, an *s-local strategy* is a mapping $\Lambda_s : Q \to 2^{Q \times (\{\Box,\Diamond\} \times 2^A) \cup \{\varepsilon\}}$ from states to sets of atoms. An s-local strategy Λ_s is *valid* if and only if it agrees with the transition function δ, that is, if and only if $\Lambda_s(q)$ satisfies $\delta(q, s)$ for all $q \in Q$. From a position (q, s) in the acceptance game, an s-local strategy defines the possible infinite sequences of ε-transitions and the possible finite sequences of ε-transitions followed by some non-ε-atom.

On infinite sequences, player *accept* loses the acceptance game if and only if the highest color that occurs infinitely often is odd. We say that an s-local strategy Λ_s is *q-losing* for player *accept* if and only if some ε-cycle with odd maximal color is reachable from q; that is, if there is a sequence $q_1 q_2 \dots q_m q'_1\ q'_2 \dots q'_n q'_1$ starting in $q_1 = q$, with $\forall i = 1 \leq i < m.(q_{i+1}, \varepsilon) \in \Lambda_s(q_i)$, $(q'_1, \varepsilon) \in \Lambda_s(q_m)$, $\forall i = 1 \leq i < n.(q'_{i+1}, \varepsilon) \in \Lambda_s(q'_i)$, and $(q'_1, \varepsilon) \in \Lambda_s(q'_n)$ such that the highest color of $q'_1, \dots, q'_n$ is odd.

On finite sequences, we are interested in the next non-ε-atom and in the highest color that was encountered during the sequence. We say that, for a color $c \in C$, an atom $(q, \circ, A') \in Q \times \{\Box, \Diamond\} \times 2^A$ is a *c-successor* of a state q_1 in an s-local strategy Λ_s if there is a sequence $q_1 q_2 \dots q_n q$ starting in q_1 such that $\forall i = 1 \leq i < n.(q_{i+1}, \varepsilon) \in \Lambda_s(q_i)$, $(q, \circ, A') \in \Lambda_s(q_n)$ and the highest color of $q_1, \dots, q_n$ is c.

Both attributes of local strategies are easy to check, because the existence of some sequence with the required property implies the existence of such a sequence with length $\leq 2|Q|$.

Lemma 4.2 *Given an ACG* $\mathcal{G}^\varepsilon = (\Sigma, Q, q_0, \delta, \alpha)$ *with* $\alpha(Q) = C$ *and* n *atoms, we can construct an* ε*-free ACG* $\mathcal{G} = (\Sigma, Q \times C, q_0', \delta', \alpha')$ *and at most* $n \cdot |C|$ *atoms that accepts the same language.*

Construction: Let L_σ^q denote the set of valid s-local strategies for a state $s \in S$ labeled with σ ($l(s) = \sigma$) that are not q-losing for player *accept*. Let $\mathbf{A}_\Lambda = \{((q', c'), (\circ, A')) \in (Q \times C) \times (\{\Box, \Diamond\} \times 2^A) \mid (q', \circ, A') \text{ is a } c'\text{-successor of } q \text{ in } \Lambda\}$, and let φ_Λ denote the conjunction over the atoms in $\mathbf{A}_\Lambda$. We construct the ε-free ACG $\mathcal{G} = (\Sigma, Q \times C, q_0', \delta', \alpha')$, where

$$q_0' = (q_0, \alpha(q_0)),\ \delta' : ((q, c), \sigma) \mapsto \bigvee_{\Lambda \in L_\sigma^q} \varphi_\Lambda, \text{ and } \alpha' : (q, c) \mapsto c.$$

Proof: $\mathcal{L}(\mathcal{G}^\varepsilon) \subseteq \mathcal{L}(\mathcal{G})$: Suppose the CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is accepted by $\mathcal{G}^\varepsilon$. The memoryless winning strategy of player *accept* defines an s-local strategy Λ_s for each state $s \in S$ of $\mathcal{C}$.

In the acceptance game of $\mathcal{G}$, player *accept* has the following winning strategy:

- In a position (q, s), player *accept* chooses, in the first stage of the round, the set $\mathbf{A}_{\Lambda_s}$ of atoms.
- If the outcome of the first stage is an atom $((q', c'), (\circ, A'))$, player *accept* proceeds in the second stage as in the acceptance game for $\mathcal{G}^\varepsilon$ from the atom $(q', \circ, A')$.

$\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{G}^\varepsilon)$: Suppose the CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is accepted by $\mathcal{G}$. The memoryless winning strategy of player *accept* defines, for each position $((q, c), s)$, a set $\mathbf{A}_{\Lambda^{(q,c,s)}}$ of successors for some $\Lambda^{(q,c,s)} \in L_{l(s)}^q$. We choose such a $\Lambda^{(q,c,s)}$ for each position $((q, c), s)$ of the acceptance game.

In the acceptance game of $\mathcal{G}^\varepsilon$, player *accept* has the following *memoryful* winning strategy:

- The memory contains a local strategy, which is to be executed up to the next non-ε-transition, and the highest color that has occurred since the last non-ε-transition. Initially, the memory contains the pair $(\Lambda^{(q_0, c(q_0), s_0)}, c(q_0))$.
- Every time a non-ε-transition to a state (q', s') is executed, the memory is updated from $(\Lambda^{(q,c,s)}, c')$ to $(\Lambda^{(q', c', s')}, c(q))$.
- Every time an ε-transition (q', ε) is executed, the memorized color is updated from c' to $\max\{c', c(q')\}$.

- In the first stage of each round, player *accept* follows the stored local strategy.
- If the outcome of the first stage is an atom $(q', \circ, A')$, player *accept* proceeds in the second stage as in the acceptance game for $\mathcal{G}$ from the atom $((q', c'), (\circ, A'))$, where c' denotes the memorized color. □

4.4 Bounded Models

We now show that, for every ACG $\mathcal{G}$, there exists a bound m such that $\mathcal{G}$ is empty if and only if $\mathcal{G}$ does not accept any m-bounded CGSs.

Given some CGS $\mathcal{C}$ in the language of $\mathcal{G}$, we transform $\mathcal{C}$ into an m-bounded CGS $\mathcal{C}'$ that is also accepted by $\mathcal{G}$. This bound m depends only on the size of $\mathcal{G}$ (and not on the CGS $\mathcal{C}$ we started with).

The proof that $\mathcal{G}$ must furthermore accept some finite CGS relies on the fact that non-empty automata over finitely branching structures always accept some finite structure [Rab72, MS95]. Once the bound m is established, we can construct an automaton over m-bounded CGSs that accepts the m-bounded CGSs in the language of $\mathcal{G}$; since its language is non-empty as well, there must exist a finite CGS accepted by both automata.

Consider an ε-free ACG $\mathcal{G}$ and a CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ accepted by $\mathcal{G}$. In the following, we define a finite set Γ of decisions and a transition function $\tau' : S \times \Gamma^A \to S$, such that the resulting bounded CGS $\mathcal{C}' = (\Pi, A, S, s_0, l, \Gamma, \tau')$ is also accepted by $\mathcal{G}$. Before we formally define the construction in the proof of Theorem 4.3 below, we first give an informal outline.

Let us begin with the special case where all atoms of $\mathcal{G}$ are of the form $(q, \Box, \{a\})$, that is, a *universal* atom with a *single* agent. We use the set of atoms as the new set of decisions of each agent. The new transition function is obtained by first mapping the decision of each agent in $\mathcal{C}'$ to a decision in $\mathcal{C}$, and then applying the old transition function.

To map the decisions, we fix a memoryless winning strategy for player *accept* in the acceptance game for $\mathcal{C}$. After an atom $(q, \Box, \{a\})$ has been chosen in the first stage of the acceptance game, player *accept* begins the second stage by selecting a decision d_a for agent a. We map each decision $(q, \Box, \{a\})$ in $\mathcal{C}'$ to this decision d_a in $\mathcal{C}$.

Player *accept* wins the acceptance game for $\mathcal{C}'$ with the following strategy: In the first stage of each move, we apply the winning strategy of player *accept*

in the acceptance game for $\mathcal{C}$. In the second stage, we simply select the atom $(q', \Box, \{a'\})$ that was chosen in the first stage as the decision for agent a'. Since the strategy for $\mathcal{C}$ wins for all possible decisions of the agents in $A \setminus \{a'\}$, it wins in particular for the decisions selected in the transition function.

Suppose next that we still have only *universal* atoms $(q, \Box, A')$, but that the set A' of agents is not required to be singleton. There is no guarantee that the decisions of the agents in A' are consistent: An agent a may choose an atom $(q, \Box, A')$ where A' does not contain a or contains some other agent a' who made a different decision. For the purpose of computing the transition function, we therefore *harmonize* the decisions by replacing, in such cases, the decision of agent a with a fixed decision $(q_0, \Box, \{a\})$.

To win the acceptance game for $\mathcal{C}'$, player *accept* selects, after an atom $(q, \Box, A')$ has been chosen in the first stage, this atom $(q, \Box, A')$ for all agents in A'. The selection is therefore consistent for all agents in A'. Since the strategy wins for all decisions of the agents in $A \setminus A'$, it does not matter if some of their decisions have been replaced. Note that, this way, only decisions of player *reject* are changed in the harmonization.

Finally, suppose that $\mathcal{G}$ contains *existential* atoms. If an existential atom $(q, \Diamond, A')$ is the outcome of the first stage of the acceptance game, player *accept* only decides *after* the decisions of the agents in $A \setminus A'$ have been made by player *reject*. To implement this order of the choices in the computation of the transition function, we allow the player who chooses the last existential atom to override all decisions for existential atoms of his opponent. We add the natural numbers $\leq |A|$ as an additional component to the decisions of the agents. For a given combined decision of the agents, the sum over the numbers in the decisions of the agents, modulo $|A|$, then identifies one *favored* agent $a_0 \in A$. In this way, whichever player chooses *last* can determine the favored agent. Given the decision of agent a_0 for some atom $(q'', \Diamond, A'')$ or $(q'', \Box, A'')$, we replace each decision for an existential atom by an agent $a \in A \setminus A''$ by the fixed decision $(q_0, \Box, \{a\})$.

To win the acceptance game for $\mathcal{C}'$, the strategy for player *accept* makes the following choice after an atom $(q', \Diamond, A')$ has been chosen in the first stage and player *reject* has made the decisions for all agents in $A \setminus A'$: For all agents in A', she selects the atom $(q', \Diamond, A')$, combined with some number which ensures that the favored agent a_0 is in A'.

Example. Consider again the CGS $\mathcal{C}_0$, which is accepted by the ACG $\mathcal{G}_\psi$ with the winning strategy for player *accept* described in Section 4.3.1. Our

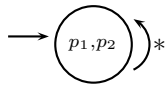


Figure 4.2: Transforming the concurrent game structure $\mathcal{C}_0$ from Figure 4.1 with the winning strategy for the automaton $\mathcal{G}_\psi$ from the example of Section 4.3 results in the depicted concurrent game structure with only a single state. Every pair (d_1, d_2) of decisions of the two participating agents from the new set $\Gamma = atom(\mathcal{G}_\psi) \times \{1, 2\}$ of decisions $(atom(\mathcal{G}_\psi) = \{(q_0, \Diamond, \emptyset), (q_{p_2}, \Diamond, \{a_2\}), (q_\nu, \Box, \{a_1\}), (q_\mu, \Box, \{a_1\})\})$ is mapped to the pair $(0, 0)$ of decisions from the old set $\Delta = \mathbb{R}$ of decisions.

construction reduces the set of decisions to the finite set $\Gamma = atom(\mathcal{G}_\psi) \times \{1, 2\} \subset \{q_0, q_\mu, q_\nu, q_{p_2}\} \times \{\Box, \Diamond\} \times 2^{\{a_1, a_2\}} \times \{1, 2\}$. The new transition function consists of two steps:

In the first step, we harmonize the given combined decision of the agents by replacing the inconsistent decisions. In the acceptance game, this may change the decisions of the agents controlled by player *reject*. If, for example, the atom $(q_{p_2}, \Diamond, \{a_2\})$ is the outcome of the first stage of the acceptance game and player *reject* makes the decision $(q_0, \Diamond, \emptyset, 1)$ for agent a_1, player *accept* responds by making the decision $(q_{p_2}, \Diamond, \{a_2\}, 1)$ for agent a_2. The sum of the natural numbers $(1+1)$ identifies agent a_2, and all existential choices for groups of agents *not* containing a_2 are overridden. The resulting choices are $(q_0, \Box, \{a_1\})$ for agent a_1 and $(q_{p_2}, \Diamond, \{a_2\})$ for agent a_2.

In the second step, the decisions of the agents are mapped to decisions in the CGS $\mathcal{C}_0$. First, the universal choices are evaluated: The winning strategy maps $(q_0, \Box, \{a_1\})$ to the decision $d_1 = 0$ for agent a_1. Then, the existential choice is evaluated: The winning strategy maps $(q_{p_2}, \Diamond, \{a_2\})$ and the decision $d_1 = 0$ for agent a_1 to the decision $d_2 = d_1 = 0$ for agent a_2.

The resulting bounded CGS is very simple: The new transition function maps all decisions to state 0 (cf. Figure 4.2).

Theorem 4.3 *An ε-free ACG $\mathcal{G} = (\Sigma, Q, q_0, \delta, \alpha)$ is non-empty if and only if it accepts an $(|atom(\mathcal{G})| \cdot |A|)$-bounded CGS.*

Proof: If $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is accepted by the ε-free ACG $\mathcal{G} = (2^\Pi, Q, q_0, \delta, \alpha)$, then player *accept* has a memoryless winning strategy in the acceptance game for $\mathcal{C}$. We fix such a memoryless strategy and use it to construct the bounded CGS $\mathcal{C}' = (\Pi, A, S, s_0, l, \Gamma, \tau')$.

Decisions. For convenience, we assume that the set A of agents is an initial sequence of the natural numbers. The new set of decisions $\Gamma = \mathit{atom}(\mathcal{G}) \times A$ consists of pairs of atoms and numbers. If the first component is an existential atom $(q, \Diamond, A')$, then the sum of the second components of the decisions of all agents is used to validate the choice.

We say that two decisions $d_1, d_2 \in \Gamma$ are *equivalent* if they agree on their first component: $(\mathbf{a}_1, a_1') \sim (\mathbf{a}_2, a_2') :\Leftrightarrow \mathbf{a}_1 = \mathbf{a}_2$.

We say that a combined decision $\upsilon \in \Gamma^A$ *favors* an agent $a \in A$, $\upsilon \rightarrowtail a$, if the sum of the second arguments, modulo $|A|$, of this combined decision is equal to a.

We say that the decision $d_a \in \Gamma$ of agent a *prevails* in the combined decision $\upsilon \in \Gamma^A$ if the following conditions hold for $d_a = ((q, \circ, A'), a'')$, $\circ \in \{\Box, \Diamond\}$:

- $a \in A'$,
- all agents $a' \in A'$ have made a decision $d_{a'} \sim d_a$ equivalent to the decision of a, and
- if $\circ = \Diamond$, then a cooperates with the agent favored by the combined decision υ ($\upsilon \rightarrowtail a' \in A'$).

Harmonization. Let $\mathbf{A} = Q \times \{\Box, \Diamond\} \times 2^A$. The harmonization $h : \Gamma^A \to \mathbf{A}^A$ maps the decision of the agents to a harmonic decision. Intuitively, a harmonic decision is a combined decision where the individual decision of every agent prevails. As the second component of Γ is not needed in this case, it can be pruned. However, given a decision $d \in \Gamma$, there is no guarantee that we can change the non-prevailing choices consistently within the set Γ^A such that all individual decisions prevail. We therefore lift the restriction to atoms of $\mathcal{G}$: Harmonic decisions are elements of $\mathbf{A}^A$ such that

- each agent $a \in A$ chooses an atom $(q, \circ, A')$ with $q \in Q$, $\circ \in \{\Box, \Diamond\}$, and $a \in A' \subseteq A$,
- if an agent $a \in A$ chooses an atom $(q, \circ, A') \in \mathbf{A}$, then all agents $a' \in A'$ choose the same atom, and
- if an agent $a \in A$ chooses an existential atom $(q, \Diamond, A') \in \mathbf{A}$, then all agents $a' \notin A'$ choose universal atoms.

For prevailing decisions, the harmonization h only deletes the second component. Non-prevailing decisions of an agent a are replaced by the fixed decision $(q_0, \Box, \{a\})$ (which is not necessarily in $\mathit{atom}(\mathcal{G})$).

Direction. We define the function $f_s : \mathbf{A}^A \to \Delta^A$ that maps a harmonic decision to a direction $\upsilon \in \Delta^A$ in $\mathcal{C}$. f_s depends on the state $s \in S$ of $\mathcal{C}$ and is inferred from the second stage of the fixed memoryless strategy. (Note that fresh atoms $(q_0, \Box, \{a\})$ may not be covered by this strategy. They can be mapped to an arbitrary (but fixed) decision $d_0 \in \Delta$.)

First, the universal decisions are evaluated: If an agent makes the harmonic decision $(q, \Box, A')$, then $\upsilon' \in \Delta^{A'}$ is determined by the choice of player *accept* in the second stage of the winning strategy in state s, when confronted with the atom $(q, \Box, A')$.

Then, the existential decisions are evaluated: If an agent makes the harmonic decision $(q, \Diamond, A')$ then $\upsilon' \in \Delta^{A'}$ is determined by the choice of player *accept* in the second stage of the winning strategy in state s, when confronted with the atom $(q, \Diamond, A')$ and the decision $\upsilon'' \in \Delta^{A \setminus A'}$ fixed by the evaluation of the universal harmonic decisions.

The new transition function $\tau' : S \times \Gamma^A \to S$ is defined as $\tau' : (s, \upsilon) \mapsto \tau(s, f_s(h(\upsilon)))$.

Acceptance. In the acceptance game for $\mathcal{C}'$, player *accept* has the following strategy: In the first stage of each round, she applies the winning strategy of the acceptance game for $\mathcal{C}$. The strategy for the second stage depends on the outcome of the first stage:

- If an atom $(q, \Box, A')$ is chosen in the first stage, player *accept* fixes the prevailing decision $((q, \Box, A'), 1)$ for all agents $a \in A'$.
- If an atom $(q, \Diamond, A')$ with $A' \neq \emptyset$ is chosen in the first stage and player *reject* has made the decisions d_a for all agents $a \notin A'$, player *accept* fixes the prevailing decisions $((q, \Diamond, A'), n_a)$ for the agents $a \in A'$ such that an agent $a' \in A'$ is favored.
- If an atom $(q, \Diamond, \emptyset)$ is chosen in the first stage, then player *accept* does not participate in the second stage.

We now show that the run tree $\langle R', r' \rangle$ defined by this strategy is accepting. Let $\langle R, r \rangle$ be the run tree defined by the winning strategy in the acceptance game for $\mathcal{C}$. In the following, we argue that, for each branch labeled $(q_0, s_0)(q_1, s_1)(q_2, s_2) \ldots$ in $\langle R', r' \rangle$, there is an identically labeled branch in $\langle R, r \rangle$. Since all branches of $\langle R, r \rangle$ satisfy the parity condition, $\langle R', r' \rangle$ must be accepting as well.

The root of both run trees is labeled by (q_0, s_0). If a node labeled (q_i, s_i) in $\langle R', r' \rangle$ has a child labeled (q_{i+1}, s_{i+1}), then there must be an atom $(q_{i+1}, \circ, A') \in$

$Q \times \{\Box, \Diamond\} \times 2^A$ in the set of atoms chosen by player *accept*, such that following holds: For the decision $\upsilon' \in \Gamma^{A'}$ defined by the strategy of player *accept*, there is a decision $\upsilon'' \in \Gamma^{A \setminus A'}$ such that $s_{i+1} = \tau'(s_i, (\upsilon', \upsilon'')) = \tau(s_i, f_{s_i}(h(\upsilon', \upsilon'')))$.

Now consider a node labeled (q_i, s_i) in $\langle R, r \rangle$. Since the strategy of player *accept* in the first stage of each round is identical for the two acceptance games, the atom $(q_{i+1}, \circ, A')$ is also included in the set of atoms chosen by player *accept* in the acceptance game for $\mathcal{C}$. Player *reject* can enforce the decision $\upsilon = f_{s_i}(h(\upsilon', \upsilon''))$ as follows:

- If $\circ = \Box$, player *accept* chooses the $\Delta^{A'}$ part of υ under the fixed memoryless strategy for the acceptance game of $\mathcal{C}$, and player *reject* can respond by choosing the $\Delta^{A \setminus A'}$ part of υ.

- If $\circ = \Diamond$, player *reject* can choose the $\Delta^{A \setminus A'}$ part of υ, and player *accept* will react by choosing the $\Delta^{A'}$ part of υ under the fixed memoryless strategy for the acceptance game of $\mathcal{C}$, guaranteeing that $\upsilon = f_{s_i}(h(\upsilon', \upsilon''))$.

In both cases, the new state $s_{i+1} = \tau(s_i, \upsilon)$ is chosen. The node labeled (q_i, s_i) in $\langle R, r \rangle$ must therefore have a child labeled (q_{i+1}, s_{i+1}). □

4.5 Satisfiability and Complexity

An ATμC formula is satisfiable if and only if the language of its ACG is nonempty. A simple procedure for deciding emptiness of ACGs is immediately suggested by Theorem 4.3: Since we can restrict our attention to m-bounded CGSs with fixed $m = |atom(\mathcal{G})| \cdot |A|$, we can replace $(q, (\Box, A'))$ and $(q, (\Diamond, A'))$ by the corresponding positive Boolean combinations: The resulting automaton accepts exactly the m-bounded concurrent game structures in the language of $\mathcal{G}$. To decide emptiness, we nondeterminize the automaton [MS95] and then solve the emptiness game. The complexity of this construction is double-exponential: Solving the emptiness game of the nondeterministic automaton is exponential in the number of directions, which is already exponential in the number of agents ($m^{|A|}$).

We now describe an alternative algorithm with only single-exponential complexity. Instead of going through an alternating automaton to a nondeterministic automaton, we go through a universal automaton to a *deterministic* automaton. The advantage of solving the emptiness game for a deterministic automaton instead of for a nondeterministic automaton is that the set of atoms chosen by

player *accept* is uniquely determined by the input letter; this reduces the number of choices from exponential in the number of directions to linear in the size of the input alphabet.

The construction of the universal automaton is based on the observation that the winning strategy of player *accept* that we defined in the previous section can be represented by assigning a function $f_s : Q \to 2^{atom(\mathcal{G})}$ to each state s of $\mathcal{C}$. The set $f_s(q)$ contains the atoms that player *accept* chooses in the first stage of the game at position (q, s). Since the strategy for the second stage depends only on the chosen atom and not on the states of $\mathcal{C}$ and $\mathcal{G}$, f_s determines the entire strategy of player *accept*.

The universal automaton runs on bounded CGSs that are annotated by this function f_s; that is, we extend the alphabet from Σ to $\Sigma \times (Q \to 2^{atom(\mathcal{G})})$ and a CGS is accepted if and only if f_s identifies a winning strategy for player *accept* in the acceptance game of the ACG. The construction does not change the set of states and increases the input alphabet by an exponential factor in the number of states and atoms of the ACG.

Lemma 4.4 *Given an ε-free ACG $\mathcal{G} = (\Sigma, Q, q_0, \delta, \alpha)$ and a set A of agents, we can construct a universal parity automaton $\mathcal{U} = (\Sigma \times (Q \to 2^{atom(\mathcal{G})}), (atom(\mathcal{G}) \times A)^A, Q, q_0, \delta', \alpha)$ on $\Sigma \times (Q \to 2^{atom(\mathcal{G})})$-labeled CGSs with the set $atom(\mathcal{G}) \times A$ of decisions such that $\mathcal{U}$ has the following properties:*

- *If $\mathcal{U}$ accepts a CGS $\mathcal{C} = (\Pi, A, S, s_0, l \times strat_1, atom(\mathcal{G}) \times A, \tau)$ then $\mathcal{G}$ accepts its Σ projection $\mathcal{C}' = (\Pi, A, S, s_0, l, atom(\mathcal{G}) \times A, \tau)$.*
- *If $\mathcal{U}$ is empty, then $\mathcal{G}$ is empty.*

Proof: We denote with $strat_2$ the function that maps each atom $\mathbf{a}$ of $\mathcal{G}$ to the set $D \subseteq (atom(\mathcal{G}) \times A)^A$ of decisions that are the outcome of the second stage of the acceptance game for some strategy of player reject, when the outcome of the first stage is $\mathbf{a}$ and player *accept* follows the simple strategy for the second stage described in the proof of Theorem 4.3. Generalizing $strat_2$ to sets of atoms, we define the transition function δ' of $\mathcal{U}$ by setting $\delta'(q; \sigma, s)$ to *false* if $s(q)$ does not satisfy $\delta(q, \sigma)$, and to a conjunction over $strat_2(s(q))$ otherwise.

If $\mathcal{U}$ accepts a CGS $\mathcal{C} = (\Pi, A, S, s_0, l \times strat_1, atom(\mathcal{G}) \times A, \tau)$, then player *accept* has a winning strategy for $\mathcal{C}' = (\Pi, A, S, s_0, l, atom(\mathcal{G}) \times A, \tau)$ in the acceptance game of $\mathcal{G}$, where the strategy in the first stage is defined by $strat_1$ and the strategy in the second stage is as defined in the proof of Theorem 4.3.

If $\mathcal{G}$ accepts a CGS $\mathcal{C}$, then there exists, as described in the proof of Theorem 4.3, a CGS $\mathcal{C}' = (\Pi, A, S, s_0, l, atom(\mathcal{G}) \times A, \tau)$, such that player

accept wins the acceptance game using some memoryless strategy $strat_1$ in the first stage and the canonical strategy in the second stage. The CGS $\mathcal{C}'' = (\Pi, A, S, s_0, l \times strat_1, atom(\mathcal{G}) \times A, \tau)$ is accepted by $\mathcal{U}$. □

We transform the universal parity automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ into a deterministic parity automaton by first transforming $\mathcal{U}$ into a universal Co-Büchi automaton with $O(c \cdot n)$ states and then using Safra's construction [Saf88, Pit06].

Lemma 4.5 *Given a universal automaton $\mathcal{U}$ with n states and c colors, we can construct an equivalent deterministic automaton $\mathcal{D}$ with $n^{O(c \cdot n)}$ states and $O(c \cdot n)$ colors.* □

Remark. The same construction can be used to nondeterminize alternating automata $\mathcal{A}$: Like in the construction of Lemma 4.4, we can enrich the input to $\mathcal{A}$ by a memoryless strategy, resulting in a universal automaton $\mathcal{U}$ with the same set of states and the same coloring function. Translating $\mathcal{U}$ by Lemma 4.5 to an equivalent deterministic automaton $\mathcal{D}$ and then projecting away the strategy (cf. Lemma 6.8) results in a nondeterministic automaton with the same states as $\mathcal{D}$ and the same language as $\mathcal{A}$.

Corollary 4.6 *Given an alternating automaton $\mathcal{A}$ with n states and c colors, we can construct an equivalent nondeterministic automaton $\mathcal{N}$ with $n^{O(c \cdot n)}$ states and $O(c \cdot n)$ colors.* □

Our transformation of the ACG to the deterministic automaton $\mathcal{D}$ thus increases both the number of states and the size of the input alphabet by a factor at most exponential in the number of states of the ACG. In general, the emptiness game of nondeterministic automata can be solved in time polynomial in both in the number of states, the size of the input alphabet and the transition table, where the size of the transition table is the number of disjuncts $d = \bigwedge_{\upsilon \in \Upsilon}(q_\upsilon, \upsilon)$ that appear in some disjunction $\delta(q, \sigma) = \bigvee_{i \in I} d_i$ of the transition function.

Theorem 4.7 *Given a nondeterministic parity automaton $\mathcal{N} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ with n states, t different entries in the transition table, and c colors, we can, in time $(n + t)^{O(c)}$, decide emptiness and, if $\mathcal{L}(\mathcal{N}) \neq \emptyset$, construct a finite CGS $\mathcal{C} \in \mathcal{L}(\mathcal{N})$ with at most n states in the language of $\mathcal{N}$.*

Proof: The emptiness problem can be reduced to a bipartite parity game with $n+t$ positions and c colors: Player *accept* owns the positions Q and chooses from

a position q a label $\sigma \in \Sigma$ and a disjunct from $\delta(q, \sigma)$. Player *reject* owns these disjuncts and picks state q_υ from a conjunct (q_υ, υ) of a disjunct $d = \bigwedge_{\upsilon \in \Upsilon}(q_\upsilon, \upsilon)$ (intuitively by choosing a direction $\upsilon \in \Upsilon$). The colors of the positions owned by player *accept* are defined by the coloring function α, while all states owned by player *reject* are colored by the minimum color in the mapping of α. This parity game can be solved in time $(n+t)^{O(c)}$ by Theorem 2.14 and Theorem 3.5.

$\mathcal{N}$ is empty if and only if player *reject* has a winning strategy, and the Σ-projection of a memoryless winning strategy for player *accept* defines a CGS in the language of $\mathcal{N}$. □

For general nondeterministic automata, the size of the transition table may be exponential in the number of directions. When applied to concurrent game structures, the number $|\Delta^A|$ of direction is always exponential in the number of colors (even for binary sets of decisions).

For a deterministic automaton $\mathcal{D}$, on the other hand, there is only one entry in the transition table for each state and input letter pair. Different from the emptiness game for general nondeterministic automata, the states of player *reject* can thus be represented as a product of states and input letters of $\mathcal{D}$ rather than using states and mappings from the directions to the states. This representation is much smaller, and allows for a tight complexity bound: The emptiness game of $\mathcal{D}$ can be solved in time polynomial only in the number of states and in the size of the input alphabet, which provides us with an exponential-time procedure for deciding emptiness of an ACG.

Corollary 4.8 *Given a deterministic parity automaton $\mathcal{D} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ with n states and c colors, we can, in time $(n \cdot |\Sigma|)^{O(c)}$, decide emptiness and, if $\mathcal{L}(\mathcal{D}) \neq \emptyset$, construct a finite CGS $\mathcal{C} \in \mathcal{L}(\mathcal{D})$ with at most n states in the language of $\mathcal{D}$.* □

Combining Theorem 4.3, Lemmata 4.4 and 4.5, and Corollary 4.8, we obtain the finite model property of automata over concurrent game structures.

Theorem 4.9 *Every non-empty ε-free ACG with n states, c colors, a atoms and a' agents accepts some finite CGS with $(a \cdot a')^{a'}$ directions and at most $n^{O(c\,n)}$ states, which can be constructed in time $2^{O(c^2\,n^3\,a)}$.* □

Combining Theorem 4.9 with Theorem 4.1 and Lemma 4.2, we furthermore obtain the finite model property for the alternating-time μ-calculus:

Theorem 4.10 *Given an ATμC formula φ with alternation depth d, n subformulas, and a agents, we can decide satisfiability of φ and, if φ is satisfiable, construct a model of φ with at most $n^{O(n\,d^2)}$ states and $O(d\,n\,a)$ decisions in time $2^{O(d^6\,n^4)}$.* □

Matching lower bounds for the ATμC satisfiability and synthesis problems are given by the lower bounds for the classic μ-calculus [Koz83, KV00].

Corollary 4.11 *The satisfiability and synthesis problems for the alternating-time μ-calculus are EXPTIME-complete.* □

Chapter 5

ATL* Satisfiability is 2EXPTIME-complete

Abstract

The two central decision problems that arise during the design of safety critical systems are the satisfiability and the model checking problem. While model checking can only be applied after implementing the system, satisfiability checking answers the question whether a system that satisfies the specification exists. Model checking is traditionally considered to be the simpler problem: For branching-time and fixed point logics such as CTL, CTL*, ATL, and the classic and alternating time μ-calculus, the complexity of satisfiability checking is considerably higher than the model checking complexity. We show that ATL* is a notable exception of this rule: Both ATL* model checking and ATL* satisfiability checking are 2EXPTIME-complete.

5.1 Introduction

The alternating-time temporal logic ATL* [AHK02] extends the classic branching-time temporal logic CTL* [Eme90] with path quantifiers that refer to the strategic capabilities of groups of agents. An ATL* specification $\langle\langle A' \rangle\rangle \varphi$ requires that the group A' of agents can cooperate to enforce the path formula φ. When interpreted over a concurrent game structure $\mathcal{C}$, $\langle\langle A' \rangle\rangle \varphi$ holds true in a

Logic	Model Checking (Structure)	Model Checking	Satisfiability Checking
LTL	NLOGSPACE [Eme90]	PSPACE [CES86]	PSPACE [Eme90]
CTL	NLOGSPACE [KVW00]	PTIME [CES86]	EXPTIME [CE82]
CTL*	NLOGSPACE [KVW00]	PSPACE [CES86]	2EXPTIME [Eme90]
ATL	PTIME [AHK02]	PTIME [AHK02]	EXPTIME [WLWW06]
ATL*	PTIME [AHK02]	2EXPTIME [AHK02]	**2EXPTIME**

Figure 5.1: For all previously considered branching-time temporal specifications, satisfiability checking is at least exponentially harder than model checking (in the specification). We show that ATL* is an interesting exception to this rule.

state s of $\mathcal{C}$ if the agents in A' can win a two player game against the agents not in A'. In this game, the two groups of agents take turns in making their decisions (starting with the agents in A'), resulting in an infinite sequence $ss_1s_2\ldots$ of states of the concurrent game structure $\mathcal{C}$. The agents in A' win this game, if the infinite sequence $ss_1s_2\ldots$ satisfies the path formula φ.

Since ATL* specifications can canonically be transformed into alternating-time μ-calculus (ATμC) formulas [dAHM01, AHK02], ATL* inherits the decidability and finite model property from ATμC (cf. Corollary 4.11). This translation from ATL* to ATμC comprises a doubly exponential blow-up, which is in line with the doubly exponential model checking complexity of ATL* [dAHM01, AHK02]. The *complexity* of the ATL* satisfiability and synthesis problem, on the other hand, has been an interesting open challenge since its introduction [WLWW06]: The complexity of the satisfiability problem is EXPTIME-complete for the least expressive alternating-time logic ATL [vD03, WLWW06] as well as for the most expressive alternating-time logic ATμC (Corollary 4.11), but the results of Chapter 4 only imply – together with the doubly exponential translation to ATμC [dAHM01, AHK02] – a *triply* exponential upper bound, while 2EXPTIME hardness is inherited from the syntactic sublogic CTL*, leaving an exponential gap between both bounds.

Outline. In this chapter, we introduce an automata-theoretic decision procedure to demonstrate that deciding the satisfiability of an ATL* specification and, for satisfiable specifications, constructing a model of the specifications is no more expensive than model checking: Both problems are 2EXPTIME-complete in the size of the specification. To the contrary, the cost of model checking a concurrent game structure against an ATL* specification is also polynomial in the size of the concurrent game structure. While polynomial conveys the im-

pression of feasibility, the degree of this polynomial is, for known algorithms, exponential in the size of the specification [AHK02, dAHM01].

On first glance, an automata-theoretic construction based on automata over concurrent game structures does not seem to be a promising starting point for the construction of a 2EXPTIME algorithm, because synthesis procedures based on alternating automata usually shift all combinatorial difficulties to testing their non-emptiness [KV99]. Using a doubly exponential translation from ATL* through ATμC to an equivalent ACGs suffices to proof the finite model property of ATL*, but indeed leads to a triply exponential construction.

In order to show that a constructive non-emptiness test for ATL* specifications can be performed in doubly exponential time, we combine two concepts: We first show that every model can be transformed into an *explicit* model that includes a certificate of its correctness. For this special kind of model, it suffices to build an ACG that only checks the correctness of the certificate. Finally, we show that we can construct such an automaton, which is only singly exponential in the size of the specification. Together with the exponential cost of a constructive non-emptiness test of ACGs (Theorem 4.9), we can provide a 2EXPTIME synthesis algorithm for ATL* specifications that returns a model together with a correctness certificate. 2EXPTIME-completeness then follows with the respective hardness result for the syntactic sublogic CTL* [Eme90] (and even for its fragment CTL$^+$ [Wil99]) of ATL*.

5.2 ATL*

ATL* [AHK02] extends the classical branching-time logic CTL* by path quantifiers that allow for reasoning about the strategic capability of groups of agents. In this section we recapitulate the syntax and semantics of ATL*.

ATL* Syntax. ATL* contains formulas $\langle\langle A' \rangle\rangle \psi$, expressing that the group $A' \subseteq A$ of agents can enforce that the path formula ψ holds true. Formally, the state formulas (Φ) and path formulas (Ψ) of ATL* are given by the following grammar (where $p \in \Pi$ is an atomic proposition, and $A' \subseteq A$ is a set of agents).

$$\Phi \ := \ \mathit{true} \mid p \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \langle\langle A' \rangle\rangle \Psi, \text{ and}$$

$$\Psi \ := \ \Phi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \mid \bigcirc\Psi \mid \Psi \,\mathrm{U}\, \Psi.$$

Every state formula is an ATL* formula. We call an ATL* formula *basic* if and only if it starts with a path quantifier $\langle\langle A' \rangle\rangle$.

Semantics. An ATL* specification with atomic propositions Π is interpreted over a CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$. $\|\varphi\|_{\mathcal{C}} \subseteq S$ denotes the set of states where φ holds. A CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ is a *model* of a specification φ ($\mathcal{C} \models \varphi$) with atomic propositions Π if and only if φ holds in the initial state ($s_0 \in \|\varphi\|_{\mathcal{C}}$).

For each state s of $\mathcal{C}$, $path(s)$ denotes the set of all paths in $\mathcal{C}$ that originate from s, and $path(\mathcal{C}) = \bigcup\{path(s) \mid s \in S\}$ denotes the set of all paths in $\mathcal{C}$.

For a CGS $\mathcal{C}$, a *strategy* for a set $A' \subseteq A$ of agents is a mapping $f_{A'} : S^* \to \Delta^{A'}$ from finite traces to decisions of the agents in A', and a *counter strategy* is a mapping $f^c_{A \setminus A'} : S^* \times \Delta^{A'} \to \Delta^{A \setminus A'}$ from finite traces and decisions of the agents in A' to decisions of the agents in $A \setminus A'$. For a given strategy $f_{A'}$ and counter strategy $f^c_{A \setminus A'}$, the set of *plays* starting at a position s_1 is defined as

- $plays(s_1, f_{A'}) =$
  $\{s_1 s_2 s_3 \ldots \mid \forall i \geq 1\, \exists d' \in \Delta^{A \setminus A'}.\, s_{i+1} = \tau(s_i, (f_{A'}(s_1 \ldots s_i), d'))\}$, and
- $plays(s_1, f^c_{A \setminus A'}) =$
  $\{s_1 s_2 s_3 \ldots \mid \forall i \geq 1\, \exists d \in \Delta^{A'}.\, s_{i+1} = \tau(s_i, (f^c_{A \setminus A'}(s_1 \ldots s_i, d), d))\}$.

An ATL* formula is evaluated along the structure of the formula.

- Atomic propositions and Boolean connectives are interpreted as usual:
  - $\|true\|_{\mathcal{C}} = S$,
  - $\|p\|_{\mathcal{C}} = \{s \in S \mid p \in l(s)\}$,
  - $\|\varphi \wedge \psi\|_{\mathcal{C}} = \|\varphi\|_{\mathcal{C}} \cap \|\psi\|_{\mathcal{C}}$,
  - $\|\varphi \vee \psi\|_{\mathcal{C}} = \|\varphi\|_{\mathcal{C}} \cup \|\psi\|_{\mathcal{C}}$, and
  - $\|\neg\varphi\|_{\mathcal{C}} = S \setminus \|\varphi\|_{\mathcal{C}}$.
- Basic formulas $\varphi = \langle\langle A' \rangle\rangle \psi$ hold true in a state s if the agents in A' have a strategy which ensures that all plays starting in s satisfy the path formula ψ: $s \in \|\varphi\|_{\mathcal{C}} \Leftrightarrow \exists f_{A'} : S^* \to \Delta^{A'}.\, plays(s, f_{A'}) \subseteq \|\psi\|^{path}_{\mathcal{C}}$.

For a path formula φ and a CGS $\mathcal{C}$, $\|\varphi\|^{path}_{\mathcal{C}} \subseteq path(\mathcal{C})$ denotes the set of paths of $\mathcal{C}$ where φ holds. Path formulas are interpreted as follows:

- For state formulas φ, $\|\varphi\|^{path}_{\mathcal{C}} = \bigcup\{path(s) \mid s \in \|\varphi\|_{\mathcal{C}}\}$.
- Boolean connectives are interpreted as usual:
  - $\|\varphi \wedge \psi\|^{path}_{\mathcal{C}} = \|\varphi\|^{path}_{\mathcal{C}} \cap \|\psi\|^{path}_{\mathcal{C}}$,

  - $\|\varphi \vee \psi\|_{\mathcal{C}}^{path} = \|\psi\|_{\mathcal{C}}^{path} \cup \|\varphi\|_{\mathcal{C}}^{path}$, and
  - $\|\neg\varphi\|_{\mathcal{C}}^{path} = path(\mathcal{C}) \setminus \|\varphi\|_{\mathcal{C}}^{path}$.

- A path $\pi = s_1, s_2, s_3, s_4 \ldots$ satisfies $\bigcirc\varphi$ (read: next φ), if the path $s_2, s_3, s_4 \ldots$ obtained by deleting the first letter of π satisfies φ:
  $\| \bigcirc \varphi\|_{\mathcal{C}}^{path} = \{s_1, s_2, s_3, s_4 \ldots \in path(\mathcal{C}) \mid s_2, s_3, s_4 \ldots \in \|\varphi\|_{\mathcal{C}}^{path}\}$.

- A path $\pi = s_1, s_2, s_3, s_4 \ldots$ satisfies $\varphi \,\mathrm{U}\, \psi$ (read: φ until ψ), if there is a natural number $n \in \mathbb{N}$ such that

  (1) the path $s_n, s_{n+1}, s_{n+2} \ldots$ obtained by deleting the initial sequence $s_1, s_2, s_3 \ldots s_{n-1}$ of π satisfies the path formula ψ, and

  (2) for all $i < n$, the path $s_i, s_{i+1}, s_{i+2} \ldots$ obtained by deleting the initial sequence $s_1, s_2, s_3 \ldots s_{i-1}$ of π satisfies the path formula φ:

  $\|\varphi \,\mathrm{U}\, \psi\|_{\mathcal{C}}^{path} = \{s_1, s_2, s_3, s_4 \ldots \in path(\mathcal{C}) \mid$
  $\exists n \in \mathbb{N}.\, (s_n, s_{n+1}, s_{n+2} \ldots \in \|\psi\|_{\mathcal{C}}^{path} \wedge \forall i < n.\, s_i, s_{i+1}, s_{i+2} \ldots \in \|\varphi\|_{\mathcal{C}}^{path})\}$.

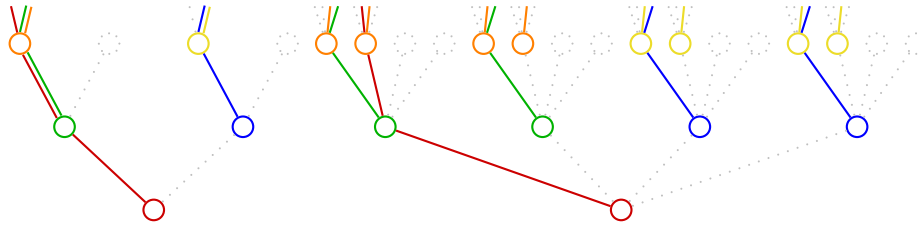
Note that the validity of basic formulas $\langle\langle A' \rangle\rangle \psi$ is implicitly defined by the outcome of a two player game with an ω-regular (LTL) objective. Such games are determined [dAHM01]. Consequently, there is a counter strategy $f_{A \setminus A'}^{c} : S^* \times \Delta^{A'} \to \Delta^{A \setminus A'}$ such that $plays(s, f_{A \setminus A'}^{c}) \subseteq \|\neg\psi\|_{\mathcal{C}}^{path}$ if and only if $s \notin \|\langle\langle A' \rangle\rangle \psi\|_{\mathcal{C}}$.

5.3 From General to Explicit Models

In this section we show that every model of a specification can be transformed into an *explicit model*, which makes both the truth of each basic subformula in the respective state and a (counter) strategy that witnesses the validity or invalidity of this basic subformulas explicit. This result is exploited in the following section by constructing a small ACG $\mathcal{A}_\varphi$ that accepts the explicit models of φ. Constructing an explicit model from a general model consists of three steps:

1. In a first step, we add a fresh atomic proposition b for each basic subformula b of φ, and extend the labeling function such that b is in the label of a states s if and only if b holds true in s ($b \in l(s) \Leftrightarrow s \in \|b\|_{\mathcal{C}}$).

   Note that witnesses for the validity or invalidity of basic subformulas cannot be encoded in the same way, because every position of the ACG may occur (multiple times) in infinitely many of these witnesses.



(a) (b)

Figure 5.2: In the central third step of the transformation of an arbitrary model into an explicit CGT, a CGT is *widened* in order to enable a finite encoding of witness strategies for the (in)validity of basic subformulas in the labels. Figure 5.2a shows a CGT for a single agent a and a binary set $\Delta = \{left, right\}$, where $\langle\langle a\rangle\rangle\varphi$ holds in every position. The color coding maps a witness strategy for $\langle\langle a\rangle\rangle\varphi$ to every position p – in the single agent case an infinite path rooted in p that satisfies φ. In Figure 5.2a, the path that always turns left is a witness strategy for the validity of $\langle\langle a\rangle\rangle\varphi$ in the root, indicated by coloring this path and the root of the tree in the same color (red). In general, witness strategies cannot be finitely encoded in the labels of a CGT, because there is no bound on the number of paths a position belongs to. The tree is therefore *widened* by extending Δ to $\Delta' = \{(left, new), (left, cont), (right, new), (right, cont)\}$ (Figure 5.2b). Witness strategies for the resulting CGT are constructed from witness strategies for the original CGT by turning first to a *new*, and henceforth to a *cont* direction, avoiding the unbounded overlap of witness strategies – for $\langle\langle a\rangle\rangle\varphi$, every position p occurs in at most one witness strategy that does not start in p – allowing for a finite representation of the witness strategies in the labels.

2. In a second step, we unravel the model obtained in the first step to a tree.

   Using trees guarantees that no position can be part of infinitely many witnesses. However, the number of witness strategies a position might belong to remains unbounded. (It may be linear in the number of its predecessors.)

3. In a final step, we *widen* the tree by adding a single Boolean decision to the set Δ of decisions available to every agent (cf. Figure 5.2).

   This widening allows us to map arbitrary but fixed witness (counter) strategies from the original tree to witness (counter) strategies in the

widened tree such that witnesses for the validity of the same basic subformula b (or its negation $\neg b$) in different states do not overlap. (With the exception of the trivial case that the witness strategy must cover all successors.) This allows us to explicitly encode the witnesses in the widened strategy trees.

From Models to Basic Models. For a given ATL* specification φ, we denote with B_φ the set of its basic subformulas. We call a model $\mathcal{C} = (\Pi \uplus B_\varphi, A, S, s_0, l, \Delta, \tau) \models \varphi$ of an ATL* formula φ *basic* if, for all basic subformulas $b \in B_\varphi$ of φ and all states $s \in S$ of $\mathcal{C}$, $b \in l(s) \Leftrightarrow s \in \|b\|_\mathcal{C}$ holds true. Since the additional propositions B_φ do not occur in the specification, the following lemma holds trivially:

Lemma 5.1 *An ATL* formula φ is satisfiable if and only if it has a basic model.* □

From Models to Tree Models. We call a CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$ a *concurrent game tree* (CGT) if $S = (\Delta^A)^*$, $s_0 = \varepsilon$, and $\tau(s, d) = s \cdot d$. For a CGS $\mathcal{C} = (\Pi, A, S, s_0, l, \Delta, \tau)$, we call $\mathcal{T}_\mathcal{C} = (\Pi, A, (\Delta^A)^*, \varepsilon, l \circ u, \Delta, \tau')$ where $\tau'(s, d) = s \cdot d$, and where the *unraveling function* $u : (\Delta^A)^* \to S$ is defined recursively by $u(\varepsilon) = s_0$, and $u(s) = s' \Rightarrow u(s \cdot d) = \tau(s', d)$, the *unraveling* of $\mathcal{C}$. We extend u to finite and infinite paths ($u(s_0 s_1 s_2 \ldots) = u(s_0)u(s_1)u(s_2)\ldots$).

Lemma 5.2 *A CGS $\mathcal{C}$ is a (basic) model of a specification φ if and only if its unraveling $\mathcal{T}_\mathcal{C}$ is a (basic) model of φ.*

Proof: By induction over the structure of φ, it is easy to prove that $s \in \|\varphi\|_{\mathcal{T}_\mathcal{C}} \Leftrightarrow u(s) \in \|\varphi\|_\mathcal{C}$, and $\pi \in \|\varphi\|^{path}_{\mathcal{T}_\mathcal{C}} \Leftrightarrow u(\pi) \in \|\varphi\|^{path}_\mathcal{C}$. The only non-trivial part in the induction is the transformation of the witness strategies for basic formulas ($\varphi = \langle\langle A' \rangle\rangle \psi$). However, we can simply use the unraveling function u to transform a witness (counter) strategy $f_{A'}$ or $f^c_{A \setminus A'}$ for $\mathcal{C}$ into a witness (counter) strategy $f'_{A'}$ or ${f^c_{A \setminus A'}}'$, respectively, for $\mathcal{T}_\mathcal{C}$. For this, we fix $f'_{A'}(\pi) = f_{A'}(u(\pi))$ or ${f^c_{A \setminus A'}}'(\pi, d) = f^c_{A \setminus A'}(u(\pi), d)$, respectively. This ensures that $plays(u(s), f_{A'}) = u(plays(s, f'_{A'})) := \{u(\pi) \mid \pi \in plays(s, f'_{A'})\}$, or $plays(u(s), f^c_{A \setminus A'}) = u(plays(s, {f^c_{A \setminus A'}}'))$. Using the induction hypothesis, we get $plays(s, f'_{A'}) \subseteq \|\psi\|^{path}_\mathcal{C}$ or $plays(s, {f^c_{A \setminus A'}}') \subseteq \|\neg\psi\|^{path}_\mathcal{C}$, respectively. □

From Tree Models to Explicit Tree Models. For a CGT $\mathcal{T} = (\Pi, A, (\Delta^A)^*, \varepsilon, l, \Delta, \tau)$, we call the CGT $\mathcal{T}_w = (\Pi, A, (\Delta'^A)^*, \varepsilon, l \circ h, \Delta', \tau')$, where $\Delta' = \Delta \times \{new, cont\}$, $h : (\Delta'^A)^* \to (\Delta^A)^*$ is a hiding function that hides the $\{new, cont\}$ part of a trace position-wise, and $\tau'(s, d) = s \cdot d$ is the usual transition function of trees, the (Boolean) *widening* of $\mathcal{T}$.

Lemma 5.3 *A CGT $\mathcal{T}$ is a (basic) model of a specification φ if and only if its (Boolean) widening $\mathcal{T}_w$ is a (basic) model of φ.*

Proof: By induction over the structure of φ. Again, the only non-trivial part is the transformation of the witness strategies for basic formulas ($\varphi = \langle\langle A' \rangle\rangle \psi$). For this part, we can use the hiding function h to transform a witness strategy $f_{A'}$ in $\mathcal{T}$ into a witness strategy $f'_{A'}$ in its widening $\mathcal{T}_w$ by choosing $f'_{A'}(\pi) = (f_{A'}(h(\pi)), *)$, where $* \in \{new, cont\}$ can be chosen arbitrarily. This ensures $plays(h(s), f_{A'}) = h(plays(s, f'_{A'})) := \{h(\pi) \mid \pi \in plays(s, f'_{A'})\}$. Using the induction hypothesis, we get $plays(s, f'_{A'}) \subseteq \|\psi\|^{path}_{\mathcal{C}}$. As in the previous lemma, we get the analogous result for the transformation of a witness counter strategy. □

Let, for a basic subformula $B_\varphi \ni b = \langle\langle A' \rangle\rangle \varphi_b$ of a specification φ, $a(b) = A'$ and $a(\neg b) = A \setminus A'$ denote the set of agents that cooperate to ensure φ_b and the set of their opponents, respectively, and let $E_\varphi = \{(b, new), (b, cont), (\neg b, new), (\neg b, cont) \mid b \in B_\varphi\}$ denote an extended set of subformulas. We call a concurrent game structure $\mathcal{C} = (\Pi \uplus B_\varphi \uplus E_\varphi, A, S, s_0, l, \Delta, \tau)$ *well-formed* if it satisfies the following requirements:

- $\forall s \in S.\, b \in l(s) \Rightarrow \exists d \in \Delta^{a(b)} \forall d' \in \Delta^{a(\neg b)}.\, (b, new) \in l(\tau(s, (d, d')))$,
- $\forall s \in S.\, (b, new) \in l(s) \Rightarrow$ $\exists d \in \Delta^{a(b)} \forall d' \in \Delta^{a(\neg b)}.\, (b, cont) \in l(\tau(s, (d, d')))$,
- $\forall s \in S.\, (b, cont) \in l(s) \Rightarrow$ $\exists d \in \Delta^{a(b)} \forall d' \in \Delta^{a(\neg b)}.\, (b, cont) \in l(\tau(s, (d, d')))$,
- $\forall s \in S.\, b \notin l(s) \Rightarrow \forall d \in \Delta^{a(b)} \exists d' \in \Delta^{a(\neg b)}.\, (\neg b, new) \in l(\tau(s, (d, d')))$,
- $\forall s \in S.\, (\neg b, new) \in l(s) \Rightarrow$ $\forall d \in \Delta^{a(b)} \exists d' \in \Delta^{a(\neg b)}.\, (\neg b, cont) \in l(\tau(s, (d, d')))$, and
- $\forall s \in S.\, (\neg b, cont) \in l(s) \Rightarrow$ $\forall d \in \Delta^{a(b)} \exists d' \in \Delta^{a(\neg b)}.\, (\neg b, cont) \in l(\tau(s, (d, d')))$.

For a basic subformula $B_\varphi \ni b = \langle\langle a(b) \rangle\rangle \varphi_b$ of φ and its negation $\neg b$, we call the set of traces $witness(s, b) = \{ss_1s_2s_3 \ldots \in path(s) \mid b \in l(s), (b, new) \in l(s_1)$ and $\forall i \geq 2.\, (b, cont) \in l(s_i)\}$ and $witness(s, \neg b) = \{ss_1s_2s_3 \ldots \in path(s) \mid b \notin l(s), (\neg b, new) \in l(s_1)$ and $\forall i \geq 2, (\neg b, cont) \in l(s_i)\}$ the explicit witnesses for b and $\neg b$ in s. $\mathcal{C}$ is called an *explicit model* of φ if the explicit witnesses are contained in the set of paths that satisfy φ_b and $\neg\varphi_b$, respectively. ($witness(s, b) \subseteq \|\varphi_b\|_{\mathcal{C}}^{path}$ and $witness(s, \neg b) \subseteq \|\neg\varphi_b\|_{\mathcal{C}}^{path}$ for all $s \in S$ and $b \in B_\varphi$.) Note that explicit models of φ are in particular basic models of φ.

Lemma 5.4 *Given a CGT $\mathcal{T}$ that is a basic model of an ATL* formula φ and a set of witness strategies for $\mathcal{T}$, we can construct an explicit model of φ.*

Proof: In the proof of the previous lemma, we showed that the widening $\mathcal{T}_w$ of a basic tree model $\mathcal{T}$ of φ is a basic model of φ. Moreover, we showed that, for the translation of witness (counter) strategies that demonstrate the (in)validity of a subformula $b \in B_\varphi$ of φ in a state s of $\mathcal{T}_w$, *any* extension $* \in \{new, cont\}$ can be chosen. In particular, the agents in $a(b)$ or $a(\neg b)$, respectively, can choose to first pick the *new* extension, and henceforth to pick the extension *cont*. For non-universal specifications, that is, for the case $a(b) \neq \emptyset$ or $a(\neg b) \neq \emptyset$, respectively, this particular choice provides the guarantee that states reachable under the new strategy $f'_{a(b)}$ or counter strategy ${f^c_{a(\neg b)}}'$, respectively, from different states in $\mathcal{T}_w$ are disjoint. ($\forall s_1, t_1 \in (\Delta'^A)^* \, \forall i, j > 1.\, s_1s_2s_3 \ldots \in plays(s_1, f'_{a(b)}) \wedge t_1t_2t_3 \ldots \in plays(t_1, f'_{a(b)}) \wedge s_i = t_j \Rightarrow s_1 = t_1$, and the analogous result for ${f^c_{a(\neg b)}}'$.)

For universal specifications, that is, for the case $a(b) = \emptyset$ or $a(\neg b) = \emptyset$, respectively, the respective player intuitively has no choice, and the (counter) strategy $f'_{a(b)}$ or ${f^c_{a(\neg b)}}'$ is well defined.

In both cases, we mark the positions reachable under $f'_{a(b)}$ in one step from a position s_1 with $b \in l(s_1)$ by (b, new) and positions reachable under ${f^c_{a(\neg b)}}'$ in one step from a position s_1 with $b \notin l(s_1)$ by $(\neg b, new)$, and we mark positions reachable in *more* than one step by $(b, cont)$ and $(\neg b, cont)$, respectively.

By construction, the resulting CGT $\mathcal{T}_w$ is well-formed, and $b \in l(s) \Rightarrow witness(s, b) = plays(s, f'_{a(b)})$ and $b \notin l(s) \Rightarrow witness(s, \neg b) = plays(s, {f^c_{a(\neg b)}}')$ hold. By Lemma 5.3, we also get $b \in l(s) \Rightarrow plays(s, f'_{a(b)}) \subseteq \|\varphi_b\|_{\mathcal{C}}^{path}$ and $b \notin l(s) \Rightarrow plays(s, {f^c_{a(\neg b)}}') \subseteq \|\neg\varphi_b\|_{\mathcal{C}}^{path}$. □

Theorem 5.5 *A specification has an explicit model if and only if it has a model.*

Proof: The 'if' direction is implied by the lemmata of this section. For the 'only if' direction, it is obvious that, for a given explicit model $(\Pi \uplus B_\varphi \uplus E_\varphi, A, S,$

$s_0, l, \Delta, \tau)$ of an ATL* specification φ, and for the projection of the labeling function to the atomic propositions ($l'(s) = l(s) \cap \Pi$), $(\Pi, A, S, s_0, l', \Delta, \tau)$ is a model of φ. □

5.4 ATL* Satisfiability is 2EXPTIME-Complete

We exploit the explicit model theorem by constructing an ACG $\mathcal{A}_\varphi$ from an ATL* specification φ that accepts only the explicit models of φ. Testing if a CGS is a model of φ is considerably harder than testing if it is an explicit model. The latter only comprises two simple tests: Checking the well-formedness criterion can be performed by a (safety) ACG with $O(|B_\varphi|)$ states, while, for all basic subformulas $b \in B_\varphi$ of φ, testing if all paths in $witness(s, b)$ satisfy the path formula φ_b and if all paths in $witness(s, \neg b)$ satisfy the path formula $\neg\varphi_b$ can be performed by a universal ACG that is exponential in φ_b.

Automata that check the (much weaker) model property, on the other hand, need to guarantee consistency of the automaton decisions, which is usually solved by using *deterministic* word automata to represent the single φ_b, leading to an exponentially larger ACG (with parity acceptance condition and a number of colors exponential in the length of φ).

We call a CGS $\mathcal{C}$ *plain* if all states in $\mathcal{C}$ are reachable from the initial state. We can restrict our focus without loss of generality to plain concurrent game structures, because unreachable states have no influence on the model property (nor are they traversed by an automaton).

Lemma 5.6 *For a specification φ, we can build an ACG $\mathcal{A}_w$ with $O(|E_\varphi|)$ states that accepts a plain CGS $\mathcal{C} = (\Pi \uplus B_\varphi \uplus E_\varphi, A, S, s_0, l, \Delta, \tau)$ if and only if it is well-formed.*

Proof: We can simply set $\mathcal{A}_w = (\Sigma_w, Q_w, q_0^w, \delta, \emptyset)$ with $\Sigma_w = 2^{B_\varphi \uplus E_\varphi}$ (the atomic propositions Π are not interpreted), $Q_w = \{q_0^w\} \uplus E_\varphi$, and

- $\delta(q_0^w, \sigma) = (q_0^w, \Box, \emptyset) \wedge \bigwedge_{b \in \sigma \cap B_\varphi}((b, new), \Box, a(b))$
  $\wedge \bigwedge_{b \in B_\varphi \setminus \sigma}((\neg b, new), \Diamond, a(\neg b))$
  $\wedge \bigwedge_{(b,*) \in \sigma \cap E_\varphi}((b, cont), \Box, a(b))$
  $\wedge \bigwedge_{(\neg b,*) \in \sigma \cap E_\varphi}((\neg b, cont), \Diamond, a(\neg b))$, and
- for all $e \in E_\varphi$, $\delta(e, \sigma) = true$ if $e \in \sigma$, and $\delta(e, \sigma) = false$ otherwise.

The $(q_0^w, \Box, \emptyset)$ part of the transition function guarantees that every reachable position in the input CGS is traversed, and the remainder of the transition function simply reflects the well-formedness constraints. □

Theorem 5.7 *[Eme90, KV05] Given an LTL formula φ, we can build an equivalent universal Co-Büchi word automaton whose size is exponential in the length of φ.* □

For ease of notation, the equivalent universal word automaton is read as a universal ACG $\mathcal{U}$ that accepts exactly those words that satisfy the LTL formula. (Words can be viewed as special concurrent game structures with a singleton set of decisions ($|\Delta| = 1$) or an empty set of agents ($A = \emptyset$).)

Let, for a path formula ψ, $\widehat{\psi}$ denote the formula obtained by replacing all occurrences of direct basic subformulas $b \in B_\psi$ by b (read as atomic proposition).

Lemma 5.8 *For a specification φ and every $B_\varphi \ni b = \langle\langle a(b) \rangle\rangle \varphi_b$ we can build two universal ACGs $\mathcal{A}_b$ and $\mathcal{A}_{\neg b}$ whose size is exponential in the size of $\widehat{\varphi_b}$ and that accept a plain CGS $\mathcal{C} = (\Pi \uplus B_\varphi \uplus E_\varphi, A, S, s_0, l, \Delta, \tau)$ if and only if $witness(s, b) \subseteq \|\widehat{\varphi_b}\|_{\mathcal{C}}^{path}$ and $witness(s, \neg b) \subseteq \|\neg\widehat{\varphi_b}\|_{\mathcal{C}}^{path}$, respectively, hold true for every state $s \in S$.*

Proof: By Theorem 5.7 we can translate the LTL formula $\widehat{\varphi_b}$ into an equivalent universal ACG $\mathcal{U}_b = (\Pi \uplus B_\varphi, Q_b, q_0^b, \delta_b, F_b)$ whose size is exponential in the length of $\widehat{\varphi_b}$. From $\mathcal{U}_b$, we infer the universal ACG $\mathcal{A}_b = (\Pi \uplus B_\varphi \uplus E_\varphi, Q_b \times \{new, cont\} \uplus \{q_b\}, q_b, \delta, F_b \times \{cont\})$ with the following transition function:

- $\delta(q_b, \sigma) = (q_b, \Box, \emptyset)$ if $b \notin \sigma$, and
- $\delta(q_b, \sigma) = (q_b, \Box, \emptyset) \wedge \bigwedge_{q \in \delta_b(q_0^b, \sigma)}((q, new), \Box, \emptyset)$ otherwise,
- $\delta((q, new), \sigma) = true$ if $(b, new) \notin \sigma$, and
- $\delta((q, new), \sigma) = \bigwedge_{q' \in \delta_b(q, \sigma)}((q', cont), \Box, \emptyset)$ otherwise, and
- $\delta((q, cont), \sigma) = true$ if $(b, cont) \notin \sigma$, and
- $\delta((q, cont), \sigma) = \bigwedge_{q' \in \delta_b(q, \sigma)}((q', cont), \Box, \emptyset)$ otherwise.

δ again uses the $(q_b, \Box, \emptyset)$ part of the transition function to traverse every reachable position in the input CGS. The assignments $\delta((q, *), \sigma) = true$ ensure that, starting in any reachable state s, only the infinite paths in $witness(s, b)$

are traversed. The remaining transitions reflect the requirement that, for all reachable positions s, all paths in $witness(s, b)$ must satisfy the path formula $\widehat{\varphi_b}$.

$\mathcal{A}_{\neg b}$ can be constructed analogously. □

Theorem 5.9 *For a given ATL* specification φ, we can construct an ACG $\mathcal{A}_\varphi$ that is exponential in the size of φ and that accepts a plain CGS if and only if it is an explicit model of φ.*

Proof: We build the automaton $\mathcal{A}_\varphi = (2^{\Pi \uplus B_\varphi \uplus E_\varphi}, \{q_0\} \uplus Q_w \uplus \biguplus_{b \in B_\varphi} \{q_b, q_{\neg b}\} \uplus (Q_b \uplus Q_{\neg b}) \times \{new, cont\}, q_0, \delta, \biguplus_{b \in B_\varphi} (F_b \uplus F_{\neg b}) \times \{cont\})$ that consists of the states of the ACG $\mathcal{A}_w$ and, for every basic subformula $b \in B_\varphi$ of φ, of the ACGs $\mathcal{A}_b$ and $\mathcal{A}_{\neg b}$, and a fresh initial state q_0. The transition function for the non-initial states is simply inherited from the respective ACG, and for the initial state we set $\delta(q_0, \sigma) = \mathit{false}$ if σ does not satisfy φ (when read as a Boolean formula over atomic propositions and basic subformulas), and $\delta(q_0, \sigma) = \delta(q_0^w, \sigma) \wedge \bigwedge_{b \in B_\varphi} \delta(q_b, \sigma) \wedge \delta(q_{\neg b}, \sigma)$ otherwise.

The lemmata of this section imply that $\mathcal{A}_\varphi$ is exponential in the size of φ, and accepts a plain CGS if and only if it is an explicit model of φ. □

It is only a small step from the non-emptiness preserving reduction of ATL* to a 2EXPTIME algorithm for ATL* satisfiability checking and synthesis.

Together, Theorems 4.9, 5.5 and 5.9 provide a 2EXPTIME algorithm for a constructive satisfiability test for an ATL* specification. The corresponding hardness result can be inferred from the 2EXPTIME-completeness of the satisfiability problem for the syntactic sublogic CTL* [Eme90] (and even for its fragment CTL$^+$ [Wil99]) of ATL*.

Corollary 5.10 *The ATL* satisfiability and synthesis problems are 2EXPTIME-complete.* □

5.5 Conclusions

We showed that the satisfiability and synthesis problem of ATL* specifications is 2EXPTIME-complete. This result is surprising: For the remaining branching-time temporal logics, the satisfiability problem is at least exponentially harder than the model checking problem [Eme90, KV99] (in the size of the specification). ATL* specifications thus seem to be particularly well suited for synthesis: They form one of the rare exceptions of the rule that testing is simpler than constructing a solution.

What is more, the suggested reduction indicates that ATL* synthesis may be feasible. The exponential blow-up in the construction of the ACG is the same blow-up that occurs when translating an LTL specification to a nondeterministic word automaton. While this blow-up is unavoidable in principle, it is also known that no blow-up occurs in most practical examples. This gives rise to the assumption that, for most practical ATL* specifications φ, the size of the emptiness equivalent Co-Büchi ACG $\mathcal{A}_\varphi$ will be small. Moreover, $\mathcal{A}_\varphi$ is essentially universal (plus a few simple local constraints), and can therefore be treated with the bounded synthesis techniques proposed in Chapter 7.

Part III

Open & Distributed Synthesis

Overview

In the traditional approach to distributed synthesis [PR90, MT01, KV01, WM03], linear or branching-time specification languages are used to specify the behavior of distributed systems, while the distribution is due to a physical division of the system into communicating processes, whereas alternating-time specifications cause a division by assigning different objectives to the agents.

This part of the thesis covers the classic problem of distributed synthesis, and connects it with alternating-time specification languages. We study systems with a predefined architecture, which is given as a directed graph. The nodes of this graph represent processes and may include the external environment as a special process. The edges of the graph are labeled with variables, indicating that data may be transmitted between two processes. These variables have a dual functionality: They are used for inter-process communication and, at the same time, serve as atomic propositions in the system specification.

We generalize the established architecture models [PR90, KV01, MT01]:

- Different to Pnueli and Rosner [PR90], and Madhusudan and Thiagarajan [MT01], we allow for cyclic architectures, such as rings.
- While Kupferman and Vardi [KV01] assumed that all processes present the same output to all adjunct processes (complete broadcast), Pnueli and Rosner [PR90], and Madhusudan and Thiagarajan [MT01] assume that different edges of the communication graph are labeled with disjoint sets of variables (excluded broadcast).

  The suggested architecture model allows for a graded form of broadcast, leaving the choice to the system designer.
- While previous architectural models [PR90, KV01, MT01, WM03] distinguish only between system processes and the environment, we distinguish between four different types of processes:

- We distinguish *white-box processes*, which come with a known and fixed implementation, from *black-box processes*, for which an implementation is sought. This is inspired by the fact that systems are often build in large parts from legacy products, which are not re-implemented, but rather used as plug-in components. Many problems of incomplete information that occur during the process of synthesis refer to black-box components only, and the distinction between black-box and white-box components significantly enhances the scope of automated decision procedures.
- We also distinguish *deterministic processes* from *nondeterministic processes*. From a technical point of view, introducing nondeterminism is a necessity if we allow for alternating-time specification languages. The introduction of nondeterminism also provides a deeper insight into the mechanisms that lead to undecidability.

The architecture defines the level of information each process can obtain about the current state of the overall system. Usually, this information is incomplete, resulting in the general undecidability of distributed synthesis [PR90]. In spite of this general undecidability result, many architectures, most notably pipelines [PR90], chains and rings [KV01], have decidable synthesis problems.

The results of this thesis show that the basic concept that distinguishes architectures with a decidable synthesis problem from undecidable architectures is the possibility to order the black-box processes with respect to their informedness.

For synchronous systems, we introduce *information forks*, a simple but comprehensive criterion that characterizes all architectures for which the synthesis problem is undecidable. An information fork describes a situation, where the environment – or, more general, nondeterministic processes – can send incomparable information to different black-box processes, unobservable by the other. We show that the absence of information forks can be checked by a quadratic algorithm, and provide a unified synthesis algorithm for all fork-free architectures (Chapter 6).

While Chapter 6 draws a clear line between decidable and undecidable architectures, the complexity of distributed synthesis – which is nonelementary even in the decidable fragment – seems to prohibit the transition of synthesis techniques into practice. We argue that this intuition is misleading: It relies on the fact that there are specifications and decidable architectures such that the smallest distributed implementation has a size nonelementary in the length of

the specification [Ros92]. But this effect does not vanish only because the program is constructed by a human rather than by a machine. Practical experience shows that, in most cases, small distributed models exist. What is more, small implementations are often an implicit design constraint, for example, because the size of the memory is limited. In Chapter 7, we therefore introduces the *bounded synthesis* problem, where we seek small implementations of bounded size. Different to distributed synthesis, bounded synthesis is decidable for all architectures, even if they contain an information fork. Furthermore, the complexity of bounded synthesis is nondeterministic quasilinear in the *minimal output*, rather than nonelementary in the *input*.

Additional types of processes are integrated into the automata theoretic framework in Chapters 8 and 9. *Probabilistic* processes (Chapter 8) choose their actions randomly rather than nondeterministically. Probabilistic processes are useful abstractions for unknown environments; we show that open synthesis is not harder for probabilistic environments (EXPTIME-complete and 2EXPTIME-complete for CTL and LTL specifications, respectively).

In Chapter 9, *reactive* processes that have the power to disable a subset of their actions are considered. They prove to be useful in devising a compositional synthesis rule for the construction of distributed systems. We show that the incomplete information each process has about the global system state does not increase the complexity for open synthesis with reactive environments (3EXPTIME-complete for CTL*, and 2EXPTIME-complete for CTL and the classic and alternating-time μ-calculus).

The synthesis algorithms suggested in the Chapters 6 through 9 assume the processes to run synchronously. Synthesis of asynchronous systems (Chapter 10) is more difficult: While synchronous processes are aware of each change to their inputs, asynchronous processes may fail to see certain changes (when the writing process is faster than the reading process) and may see duplicate input values (when the reading process is scheduled multiple times between two writes).

The central new aspect in asynchronous synthesis is the concept of a *scheduler* that, in each turn, decides which group of processes is scheduled. It turns out that the existence of an order on the informedness of the black-box processes remains the basic concept that determines decidability.

A scheduler can, in general, destroy any order of informedness between different processes: We show that the synthesis of asynchronous distributed systems is decidable if and only if at most one process implementation is unknown. The cost of synthesizing a single-process implementation is the same for synchronous and asynchronous systems (2EXPTIME-complete for CTL*, and EXPTIME-

complete for CTL and the classic μ-calculus) if we assume a known scheduler, such as the full scheduler that allows every possible scheduling behavior. Lifting this assumption leaves an exponentially harder synthesis problem for asynchronous systems.

For systems that are globally asynchronous, but have local synchronization quotients (GALS systems), the scheduler cannot interfere with the relative informedness between different processes from the same synchronization quotient: We show that synthesis remains decidable if and only if all black-box processes are contained in a single fork-free synchronization quotient.

Chapter 6

Uniform Distributed Synthesis

Abstract

We provide a uniform solution to the problem of synthesizing a finite-state distributed system. An instance of the synthesis problem consists of a system architecture and a temporal specification. The architecture is given as a directed graph, where the nodes represent processes that communicate synchronously through shared variables attached to the edges. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of data. A solution to the synthesis problem is a collection of finite-state programs for the processes in the architecture, such that the joint behavior of the programs satisfies the specification in an unrestricted environment. We define information forks, a comprehensive criterion that characterizes all architectures with an undecidable synthesis problem. The criterion is effective: For a given architecture with n processes and v variables, it can be determined in $O(n^2\, v)$ time whether the synthesis problem is decidable. We give a uniform synthesis algorithm for all decidable cases. Our algorithm works for all ω-regular tree specification languages, including the alternating-time μ-calculus. The undecidability proof, on the other hand, uses only LTL or, alternatively, CTL as specification language. Our results therefore hold for the entire range of specification languages from LTL/CTL to the alternating-time μ-calculus.

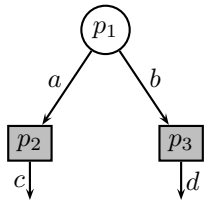


Figure 6.1: The architecture $\mathbf{A}_0$

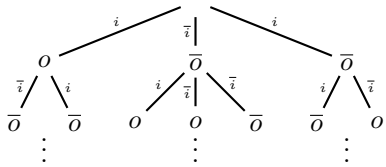
6.1 Introduction

Program synthesis, which automatically transforms a specification into a correct implementation, has been an active field of research since *Church's solvability problem* [Chu63] in the early sixties. For a given sequential specification over two sets I, O of Boolean input and output variables, Church's problem is to find an implementation $f : (2^I)^\omega \to (2^O)^\omega$ such that $(i, f(i))$ satisfies the specification for all possible input sequences $i \in (2^I)^\omega$. Church's problem has been intensively studied in the setting of temporal logics [CE82, Wol82, PR89a, KV97b, KV00].

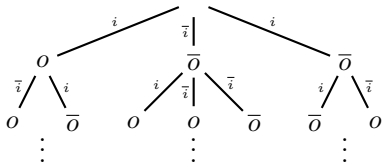
More recently, Church's problem has been extended to distributed systems [PR90, KV01, MT01, WM03], where the implementation consists of several independent processes which must choose their actions based on generally incomplete information about the system state.

The synthesis algorithms in the literature solve various instances of this problem that differ in the choice of the system architecture and the specification logic. *Closed synthesis*, the case of a single-process implementation without any interaction with the environment, was solved for CTL [CE82] and LTL [Wol82]. *Open synthesis* concerns systems consisting of a single process and an environment and was solved for LTL [PR89a] and CTL* [KV97b] as well as for the μ-calculus [KV00]. An automata-based synthesis algorithm for pipeline and ring architectures and CTL* specifications is due to Kupferman and Vardi [KV01]; Walukiewicz and Mohalik provided an alternative game-based construction [WM03]. There is also a negative result: Pnueli and Rosner [PR90] showed that the synthesis problem is undecidable for LTL specifications and the simple architecture $\mathbf{A}_0$ in Figure 6.1, consisting of an unconstrained environment and two independent system processes.

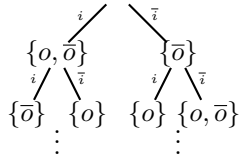
The question arises whether it is necessary to continue this series of isolated results, one for each architecture and logic. Can synthesis be extended to



(a) inconsistent strategy



(b) consistent strategy



(c) choice-set tree

Figure 6.2: Figure 6.2a shows an *inconsistent* nondeterministic strategy: On the leftmost branch (with input $i \cdot \overline{i}$) the process always reacts with output $\overline{o}$, while on the rightmost branch, with identical input $i \cdot \overline{i}$, it reacts with output o. Figure 6.2b shows a *consistent* nondeterministic strategy. Figure 6.2c shows the choice-set representation of the consistent nondeterministic strategy of Figure 6.2b.

alternating-time specifications, and thus to nondeterministic implementations? Can we provide a comprehensive criterion to determine if the distributed synthesis problem for a given system architecture and specification logic is decidable? And can the synthesis problem in fact be solved *uniformly*, that is, by a single algorithm for all decidable cases? This chapter gives a positive answer to these questions.

Nondeterministic Processes. Giving a positive answer to the first question means solving the synthesis problem for alternating-time logics. For this purpose, we generalize Church's notion of an implementation as a *deterministic strategy* or *function* $f : (2^I)^\omega \to (2^O)^\omega$ to *nondeterministic strategies* or *relations* $r \subseteq (2^I)^\omega \times (2^O)^\omega$, which allow for multiple possible outcomes due to *choices* made by the process.

Church's representation facilitates the development of automata-theoretic synthesis algorithms, because deterministic strategies can be represented as trees that branch according to the possible inputs. Each node carries a label that

indicates the output of the process after seeing the input defined by the path to the node. Sets of such trees can be represented as tree automata, and can therefore be manipulated by standard tree automata operations.

Along the same lines, nondeterministic strategies can be understood as trees that branch not only according to inputs but also to the choices of the process. However, in this representation, sets of implementations can no longer be represented by tree automata, because tree automata cannot ensure that the choices available to the process are consistent with its observations: A strategy tree is *consistent* if every pair of nodes that are reached on paths labeled by the same input allows the *same set of choices* (for each input). For example, Figure 6.2a shows an inconsistent strategy tree, while the strategy tree in Figure 6.2b is consistent. Unfortunately, the consistent trees do not form a regular language, and can therefore not in general be recognized by tree automata.

We solve this problem with a new encoding of nondeterministic strategies as trees where *each node is labeled by the set of possible choices*. Figure 6.2c shows the representation of the consistent strategy of Figure 6.2b as such a choice-set tree. Choice-set trees always represent consistent strategies, and every consistent strategy can be represented as a choice-set tree (modulo bisimilarity). Using the choice-set representation, we define an automata-theoretic synthesis algorithm which solves the distributed synthesis problem for all *hierarchical architectures*.

Uniform Synthesis Algorithm. In the *uniform distributed synthesis* problem, we decide for a given architecture $\mathbf{A}$ and a temporal specification φ over a set of Boolean variables Π whether there exists a finite-state program for each process in $\mathbf{A}$, such that the composition of the programs satisfies φ. The architecture $\mathbf{A}$ is given as a directed graph, where the nodes represent processes. The edges of the graph are labeled by variables from Π, indicating that data can be transmitted between two processes. The same variable may occur on multiple outgoing edges of a single node, allowing for the broadcast of data. Among the processes, we distinguish different types: A process is *black-box* if its implementation is unknown and needs to be discovered by the synthesis algorithm. A process is white-box if the implementation is already known and fixed. We also distinguish processes that are required (in case of black-box processes) or known (in case of white-box processes) to be deterministic from processes that may show or are known to display a nondeterministic behavior[1]. Figure 6.3 shows

[1] As comparison, in traditional approaches to distributed synthesis, the environment, which is unconstrained in the sense that it always enables all possible actions, has been the only nondeterministic process.

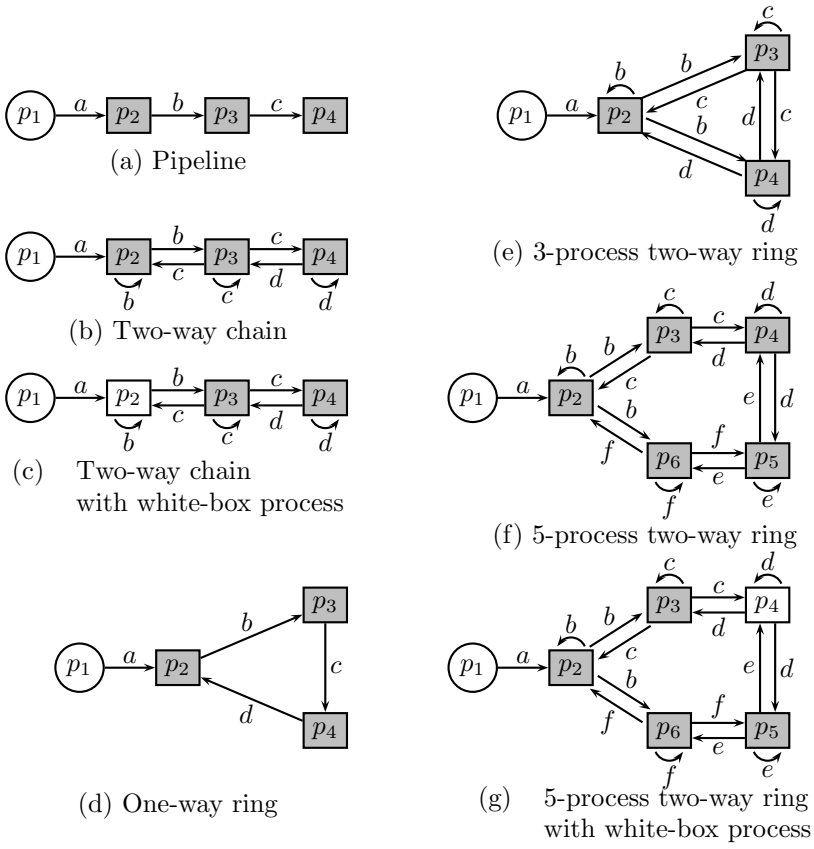


Figure 6.3: Distributed architectures

several example architectures, depicting the environment as a circle, black-box processes as filled rectangles, and white-box processes as empty rectangles.

We provide a comprehensive criterion for the decidability of the synthesis problem for a given architecture: The problem is decidable if and only if the architecture does not contain an information fork. Intuitively, an *information fork* is a situation where two black-box processes receive – directly or indirectly – information from nondeterministic processes (for example, from the environment) in such a way that they cannot completely deduce the information received by the other process.

With the information fork criterion, it is simple to determine for a given architecture whether the synthesis problem is decidable. Consider, for example, the 5-process two-way ring of Figure 6.3f. The synthesis problem is undecidable even if the white-box process p_1 (which intuitively represents an unconstrained external environment) is the only nondeterministic process, because of the information fork in the processes p_4 and p_5. The environment p_1 can transmit information through a, b, c to p_4 that remains unknown to p_5, and, vice versa, transmit information through a, b, f to p_5 that remains unknown to p_4. Interestingly, the architecture becomes decidable if we eliminate one of the two processes (resulting in a 4-process two-way ring) or, alternatively, fix the implementation of p_4 or p_5, turning the process into a white-box, as shown for p_4 in Figure 6.3g.

Different to deterministic processes, the output of nondeterministic processes cannot be inferred from their input. This leads to a slight restriction of the class of decidable architectures. Two way rings, for example, remain decidable up to the 3-process two-way ring from Figure 6.3e (while 4-process two-way rings are decidable for deterministic black-box processes). For nondeterministic processes, the two-way chain in Figure 6.3b is undecidable, because of the information fork in the processes p_2 and p_4: While the information a is unknown to process p_4, process p_2 cannot infer the value of d. The architecture becomes decidable if we turn p_2 white-box (Figure 6.3c).

The information fork criterion connects and extends the isolated decidability results in the literature. Pipelines and one-way rings, for example, have decidable synthesis problems [KV01] because the environment cannot communicate any information to a process without giving the same information to all processes to the left (when depicted as in Figure 6.3). By allowing for both broadcast and single-process communication, we distinguish the undecidable architecture $\mathbf{A}_0$ of Figure 6.1 from the decidable architecture that can be obtained by adding variable a to the edge between processes p_1 and p_3 in architecture $\mathbf{A}_0$. By identifying processes as black-box and white-box, we distinguish the decidable architecture in Figure 6.3g from the undecidable two-way ring in Figure 6.3f. And by identifying deterministic processes, we distinguish architectures that are decidable for multi-agent systems (such as the 3-process ring in Figure 6.3e) from architectures that are decidable only for deterministic processes (such as the one-way ring of Figure 6.3d) and undecidable architectures (such as the 5-process two-way ring of Figure 6.3f).

We solve the uniform synthesis problem with a single algorithm for all decidable cases. The algorithm consists of a first phase in which any architecture without an information fork is transformed into a hierarchical architecture, that is, an architecture where the input to different black-box processes is pairwise

comparable, with an equivalent synthesis problem. For this type of architecture, we solve the synthesis problem with an automata-based construction that successively eliminates processes along the information order, starting with the best-informed process.

6.2 The Synthesis Problem

In this chapter, we solve the distributed synthesis problem for the alternating-time μ-calculus. Given an ATμC formula φ and a system architecture, we decide if there exists a distributed implementation that satisfies φ.

In a distributed system where all processes cooperate, we can assume that the behavior of every process is fixed *a priori*: in each state, the next transition follows a deterministic strategy. If we allow for non-cooperating behavior, we can no longer assume a deterministic choice. Instead, we fix the set of *possible decisions* and the effect each decision has on the state of the system. At each point in a computation, the processes choose a decision from the given set and the system continues in the successor state determined by that choice.

6.2.1 Architectures

In a distributed system, it is not generally the case that every process is informed about the decisions of all other processes. The system architecture fixes a set of output variables for each process such that every decision corresponds to a certain value of the output variables. An output variable can be an input variable to another process, indicating that the value of the variable is communicated to that process. An *architecture* is a tuple $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ with

- a set A of processes, which is partitioned into a set $B \subseteq A$ of *black-box* processes, whose implementations we wish to synthesize, and a set $W = A \setminus B$ of *white-box* processes, which have known and fixed implementations,
- a set $D \subset A$ of deterministic processes,
- a set Π of system variables that also serve as atomic propositions,
- a family $\{I_a\}_{a\in A}$ of sets of input variables, such that $I_a \subseteq \Pi$ denotes the variables visible to agent a, and
- a family $\{O_a\}_{a\in A}$ of non-empty sets of output variables that disintegrates the set Π of system variables.

An architecture is called *hierarchical* if the informedness relation $\preceq = \{(b, b') \in B \times B \mid I_b \subseteq I_{b'}\}$ on the black-box processes is a linear preorder.

6.2.2 Implementations

An implementation defines for each position of a computation a subset of the output values as the set of possible decisions available to a process. The set of possible decisions must be consistent with the knowledge of the process: An *implementation* of a nondeterministic process $a \in A \setminus D$ is a function

$$p_a : (2^{I_a})^* \to 2^{2^{O_a}} \setminus \{\emptyset\} = \mathcal{O}_a$$

that assigns a choice-set of possible output values to each history of input values. For deterministic processes $a \in D$, $\mathcal{O}_a$ is restricted to singleton choice-sets:

$$p_a : (2^{I_a})^* \to \{\mathbf{O}_a \subseteq 2^{O_a} \mid |\mathbf{O}_a| = 1\} = \mathcal{O}_a.$$

Occasionally, we consider implementations that have access to a superset I of their input variables. We call a function $p_a : (2^I)^* \to \mathcal{O}_a$ with $I_a \subset I$ a *relaxed* implementation of a with input I.

A *distributed implementation* is a set $P = \{p_a\}_{a \in A}$ of process implementations, one for each process a in the architecture.

We identify process implementations with trees. As usual, an Υ-tree is a prefix closed subset $Y \subseteq \Upsilon^*$ of finite words over a predefined set Υ of directions. For a non-empty word $y \cdot \upsilon$, $x \in \Upsilon^*, \upsilon \in \Upsilon$, we call the last letter υ the *direction* $dir(y \cdot \upsilon) = \upsilon$, and fix a *root-direction* $\upsilon_0 = dir(\varepsilon) \in \Upsilon$ for the empty word ε. For given sets Σ and Υ, a Σ-*labeled* Υ-*tree* is a pair $\langle Y, l \rangle$, consisting of a tree $Y \subseteq \Upsilon^*$ and a labeling function $l : Y \to \Sigma$ that maps every node of Y to a letter of Σ. If Υ and Σ are not important or clear from the context, $\langle Y, l \rangle$ is called a tree. If $Y \ni \varepsilon$ contains the empty word and, for each $y \in Y$, some successor $y \cdot \upsilon$ ($\upsilon \in \Upsilon$) of y is in Y, then Y and $\langle Y, l \rangle$ are called *total*. If $Y = \Upsilon^*$, Y and $\langle Y, l \rangle$ are called *full*.

6.2.3 Compositions

The strategies of the processes are composed synchronously. In every step, each process $a \in A$ fixes, based on the history of inputs visible to a, a set $\mathbf{O}_a \in \mathcal{O}_a$ of possible valuations of its output variables O_a. The composition of a set $S \subseteq A$ of processes maps the global input history to the possible valuation of their joint

output variables $\bigcup_{a\in S} O_a$, which is defined by the decisions of the strategies $\{p_a\}_{a\in S}$ of the processes in S.

As an auxiliary notion, we first define the composition of sets:

- We define the composition $A \oplus B = \{a \cup b \mid a \in A, b \in B\}$ of sets of sets A and B as the set of unions of their elements.
- Likewise we define the composition $A \otimes B = \{a \oplus b \mid a \in A, b \in B\}$ of sets of sets of sets A and B as the set of compositions of their elements.

We use $\mathcal{O}_S$ to abbreviate the set $\bigotimes_{a\in S} \mathcal{O}_a$ of possible joint decisions of the processes in S. The *composition* of two implementations p_a and $p_{a'}$ of processes a and a' *with complete information* ($I_a = I_{a'} = \Pi$) is the joint implementation $p_{\{a,a'\}} : (2^\Pi)^* \to \mathcal{O}_a \otimes \mathcal{O}_{a'}$ with $p_{\{a,a'\}}(y) = p_a(y) \oplus p_{a'}(y)$ for all $y \in (2^\Pi)^*$.

In the general case of incomplete information we need to compose strategies of processes with different sets of input variables. We define two auxiliary functions: The function *hide* projects a history of variable assignments to a history for a subset of the variables (such as the visible input variables of a process). The function *wide* extends a strategy to a larger set of input variables, without changing its behavior (that is, the extended strategy does not depend on the new input variables).

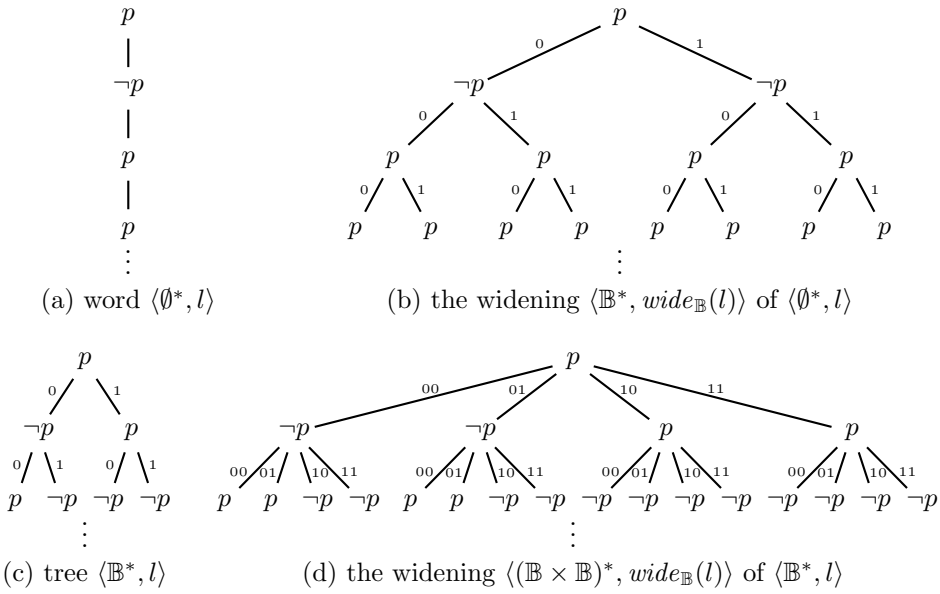
- For a set $\Xi \times \Upsilon$ of directions and a node $x \in (\Xi \times \Upsilon)^*$, $hide_\Upsilon(x)$ denotes the node in Ξ^* obtained from x by replacing (ξ, υ) by ξ in each letter of x.
- For a Σ-labeled Ξ-tree $\langle \Xi^*, l\rangle$, $\langle (\Xi \times \Upsilon)^*, wide_\Upsilon(l)\rangle$, denotes the Σ-labeled $\Xi \times \Upsilon$-tree $\langle (\Xi \times \Upsilon)^*, l'\rangle$ with $l'(x) = l(hide_\Upsilon(x))$.

The widening function $wide_\Upsilon$ guarantees that the label of each node $y \in (\Xi \times \Upsilon)^*$ in the resulting tree depends only on the visible part $hide_\Upsilon(y) \in \Xi^*$ of the input history. This construction is illustrated with an example in Figure 6.4.

We define the *composition* $p_{\{a,a'\}} : (2^\Pi)^* \to \mathcal{O}_{\{a,a'\}} = \mathcal{O}_a \otimes \mathcal{O}_{a'}$ of two strategies $p_a : (2^{I_a})^* \to \mathcal{O}_a$ and $p_{a'} : (2^{I_{a'}})^* \to \mathcal{O}_{a'}$ as the strategy $p_{\{a,a'\}} = wide_{2^{\Pi \setminus I_a}}(p_a) \oplus wide_{2^{\Pi \setminus I_{a'}}}(p_{a'})$, and use p_S as an abbreviation for the composition $\bigoplus_{a\in S} p_a$ of the strategies $\{p_a\}_{a\in S}$ of the processes in S.

6.2.4 Computations

An implementation $\{p_a : (2^{I_a})^* \to \mathcal{O}_a\}_{a\in A}$ defines the *computation tree* $\langle \|p_A\|, dir\rangle$, where $p_A = \bigoplus_{a\in A} p_a$ denotes the composition of the process strategies, and $\|p_A\|$ denotes the set of computations allowed by p_A: $\|p_A\|$ is the



(a) word $\langle \emptyset^*, l \rangle$

(b) the widening $\langle \mathbb{B}^*, \mathit{wide}_{\mathbb{B}}(l) \rangle$ of $\langle \emptyset^*, l \rangle$

(c) tree $\langle \mathbb{B}^*, l \rangle$

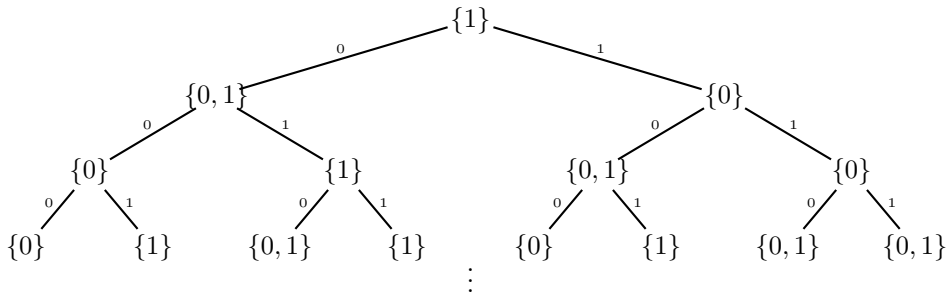
(d) the widening $\langle (\mathbb{B} \times \mathbb{B})^*, \mathit{wide}_{\mathbb{B}}(l) \rangle$ of $\langle \mathbb{B}^*, l \rangle$

Figure 6.4: The widening function wide_{Υ} maps a Ξ-tree $\langle \Xi^*, l \rangle$ to the $\Xi \times \Upsilon$-tree whose label depends only on the Ξ-part of the history. Figure 6.4a shows a unary tree (or word) as a simple input to the widening function $\mathit{wide}_{\mathbb{B}}$. The result is a $\mathbb{B}$-tree, where every path is labeled identically (Figure 6.4b). Figure 6.4d shows the result of a Boolean widening on the Boolean tree $\langle \mathbb{B}, l \rangle$ from Figure 6.4c. Here, every pair $y, y' \in (\mathbb{B} \times \mathbb{B})^*$ of nodes which are indistinguishable under hiding of the second element ($\mathit{hide}_{\mathbb{B}}(y) = \mathit{hide}_{\mathbb{B}}(y')$) has the same label $l(\mathit{hide}_{\mathbb{B}}(y))$.

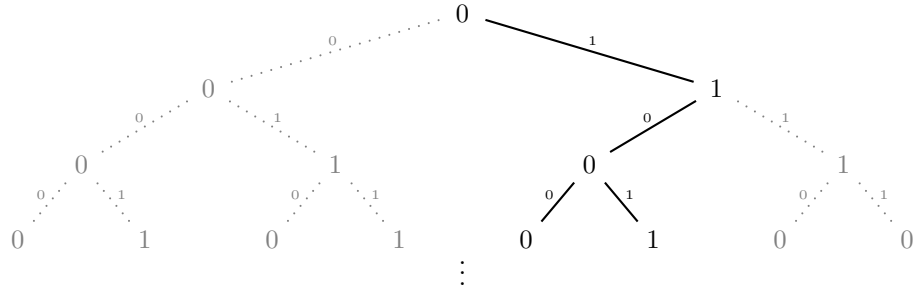
greatest total tree $Y \subseteq (2^{\Pi})^*$ such that for all $y \in (2^{\Pi})^*$ and all $\upsilon \in 2^{\Pi}$, if $y \cdot \upsilon \in Y$, then $y \in Y$ and $\upsilon \in p_A(y)$. Figure 6.5 illustrates the construction of the computation tree with an example.

Computations can be viewed as a concurrent game tree $\mathcal{G}^{\mathbf{A}}_{p_A} = (\Pi, A, S, s_0, l, \{\Delta_a\}_{a \in A}, \tau)$, where

- Π is the finite set of atomic propositions from the architecture,
- A is the set of processes,



(a) strategy tree $\langle \mathbb{B}^*, p_A : \mathbb{B}^* \to 2^{2^{\mathbb{B}}} \setminus \{\emptyset\} \rangle$



(b) computation tree $\langle \|p_A\|, dir \rangle$

Figure 6.5: Figure 6.5a shows a Boolean strategy tree $\langle \mathbb{B}^*, p_A \rangle$. Every node is labeled with a nonempty subset of $\mathbb{B}$, indicating the possible futures. Figure 6.5b shows (in solid lines) the computation tree $\langle \|p_A\|, dir \rangle$. The computation tree contains those paths of the full tree $\langle \mathbb{B}, dir \rangle$ (shown in gray) that are consistent with the strategy.

- $S = (2^{\Pi})^*$ is the set of finite sequences of valuations of the atomic propositions, with the empty sequence $s_0 = \varepsilon$ as initial state,
- $l = dir$ is a labeling function that decorates each state with its direction, and
- $\Delta_a = p_a \circ hide_{2^{\Pi \setminus I_a}}$ maps, for each process $a \in A$, a state s of the concurrent game tree to the set $\mathbf{O}_a \in \mathcal{O}_a$ of decisions that are available to a in s. $\bigoplus_{a \in A} \Delta_a$ maps each state $s \in S$ to a vector of possible decisions for the processes.

- $\tau : (s, d) \mapsto s \cdot d$ is the usual transition function for trees that maps a state s and a vector $d \in \bigoplus_{a \in A} \Delta_a(s)$ of possible decisions for the processes to a successor state $s \cdot d \in S$.

A system with composed strategy p_A satisfies a specification φ if and only if the computation tree $\langle \|p_A\|, dir \rangle \vDash \varphi$ is a model of φ.

6.2.5 Specification Languages

We consider ω-regular specification languages, in particular the alternating-time μ-calculus and the classic μ-calculus as its syntactic sublogic (where the modal operator $\Box$ is interpreted as $\Box_\emptyset$, and $\Diamond$ as $\Box_A$), and ATL* and its sublogics CTL* (where all universal and existential path quantifiers are translated to $\langle\langle\emptyset\rangle\rangle$ and $\langle\langle A\rangle\rangle$, respectively), ACTL*, LTL, and CTL.

A specification is called *universal* if it is recognized by a universal ACG. In particular, all specifications in ACTL* (and in its sublogics ACTL and LTL), and μ-calculus specifications that do not contain $\Diamond$ operators are universal.

For universal specifications, we assume that all black-box processes are deterministic: Replacing the nondeterministic implementation p_a of one or more processes by any determinization p'_a ($p'_a(y) \subseteq p_a(y) \, \forall y \in (2^I_a)^*$), the resulting computation tree $\|p'_A\| \subseteq \|p_A\|$ is a subtree of $\|p_A\|$. If a universal ACG accepts $\langle \|p_A\|, dir \rangle$, it also accepts $\langle \|p'_A\|, dir \rangle$, using the same winning strategy (cf. Subsection 9.6.5).

6.2.6 Realizability and Synthesis

The *realizability problem* is to decide for the triple $(\mathbf{A}, \varphi, \{p_w\}_{w \in W})$, consisting of an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$, an AT$\mu$C or ATL* formula φ, and a set of white-box implementations $P_W = \{p_w\}_{w \in W}$, whether there exists a set $P_B = \{p_b\}_{b \in B}$ of implementations for the black-box process in $\mathbf{A}$, such that the joint behavior $\langle \|p_A\|, dir \rangle$ satisfies φ. The *synthesis problem* is to construct such a set of implementations (provided such a set exists).

An architecture $\mathbf{A}$ is called *decidable* if realizability can be decided for all formulas φ and implementations P_W of the white-box processes. In the following sections, we give a criterion that decides if an architecture is decidable, and provide a *synthesis algorithm* that works for all decidable architectures.

6.3 Information Forks

As discussed in the introduction, an information fork is a situation where two black-box processes receive information from nondeterministic processes (directly or indirectly) in such a way that they cannot completely deduce the information received by the other process. Formally, an *information fork* is a tuple (A', p, p', q, q', E), where

- $A' \subseteq A$ is a subset of the processes,
- $p, p' \in B$ are two different black-box processes,
- $q, q' \in A'$ are two not necessarily different processes in A' such that
  - q can transmit information to p unobservable by p' – $(O_q \cap I_p) \smallsetminus I_{p'} \neq \emptyset$,
  - q' can transmit information to p' unobservable by p – $(O_{q'} \cap I_{p'}) \smallsetminus I_p \neq \emptyset$, and
  - $E \subseteq \{(a, d) \in A' \times D \cap A' \mid (O_a \cap I_d) \smallsetminus (I_p \cup I_{p'}) \neq \emptyset\}$ is a set of connections between two processes $a \in A'$ and $d \in D \cap A'$ that are unobservable by p and p', such that (A', E) is an acyclic directed graph that has only nondeterministic sources $a \notin D$[2].

For example, the architecture $\mathbf{A}_0$ contains the information fork $(\{p_1\}, p_2, p_3, p_1, p_1, \emptyset)$. The 5-process two-way ring of Figure 6.3d contains the information fork $(A', p_4, p_5, p_3, p_6, E)$ with $A' = \{p_1, p_2, p_3, p_6\}$, and $E = \{(p_1, p_2), (p_2, p_3), (p_2, p_6)\}$.

We now show that the information fork criterion is effectively decidable. Our construction is based on the observation that every architecture that does not contain an information fork can be ordered according to the relative informedness of the processes.

Consider, for a process p, the set

$$E_p = \{(a, a') \in A \times A \mid (O_a \cap I_{a'}) \smallsetminus I_p \neq \emptyset\}$$

of connections between two processes a and a' that are unobservable by p, and the set

$$U_p = \left\{ a \in A \;\middle|\; \begin{array}{l}\text{there is no nontrivial directed path from a nondeterministic} \\ \text{process } a' \in A \smallsetminus D \text{ to } a \text{ in the directed graph } (A, E_p)\end{array} \right\}$$

[2]If p or p' are deterministic, this implies $p \notin A'$ or $p' \notin A'$, respectively.

of processes that are not reachable from any nondeterministic process by such edges. The preorder $\succcurlyeq \subseteq A \times A$ (read: has more or equal information than) is then defined as follows: A process $p \in A$ has more or equal information than a process $a \in A$ if and only if a is an element of U_p ($p \succcurlyeq a \Leftrightarrow a \in U_p$). $\succeq$ is the restriction of $\succcurlyeq$ to black-box processes ($\succeq = (B \times B) \cap \succcurlyeq$).

An architecture $\mathbf{A}$ is called *ordered* if $\succeq$ is a linear preorder.

Theorem 6.1 An architecture $\mathbf{A}$ is ordered if and only if $\mathbf{A}$ does not contain an information fork.

Proof: Suppose $\mathbf{A}$ contains an information fork (A', p, p', q, q', E), then $p \not\succcurlyeq p'$ because $\big(A' \cup \{q\}, E \cup \{(q,p)\}\big)$ is a subgraph of $(A, E_{p'})$ that witnesses $p \not\succcurlyeq p'$. $p' \not\succcurlyeq p$ follows analogously.

If $\mathbf{A}$ is ordered, then $\forall p, p' \in B.\, p \succeq p' \vee p' \succeq p$. Let us assume without loss of generality that $p \succeq p'$ holds true. By definition, there is no nontrivial directed path from any nondeterministic process $a \in A \setminus D$ to p' in (A, E_p). Let us assume that (A', p, p', q, q', E) is an information fork. Then $\big(A' \cup \{q'\}, E \cup \{(q', p')\}\big)$ is a subgraph of (A, E_p), and there is a directed path from a nondeterministic process a to q' in (A', E). $\lightning$ □

Whether a given architecture $\mathbf{A}$ contains an information fork can therefore be checked by first computing $\succeq$, and then checking if each pair of black-box processes $p, p' \in B$, is comparable by $\succeq$ ($p \succeq p'$ or $p' \succeq p$).

The algorithm runs in $O(n^2 \cdot v)$ time, where $n = |A|$ is number of processes and $v = |\Pi|$ is the number of variables in the architecture $\mathbf{A}$. We show in Section 6.6 that architectures that contain an information fork are undecidable. In the following sections, we show that architectures without information forks are decidable.

6.4 Synthesis for Fork-Free Architectures

The synthesis algorithm consists of three phases: In the first phase, the architecture is transformed into a hierarchical architecture. In the second phase, an automata-based construction decides the realizability problem for the simplified architecture. If it is realizable, an implementation is constructed in the third phase.

6.4.1 Architecture Transformations

Edges from processes with a lower level of informedness to those with a higher level are always redundant, because the feedback can be simulated by the better-informed process. Feedback edges can therefore removed without changing the decidability of the architecture. Likewise, we can also add feedback edges without changing the decidability.

Let, for an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$, *feedback*$(\mathbf{A}) = (A, B, D, \Pi, \{I'_a\}_{a\in A}, \{O_a\}_{a\in A})$ denote the architecture with

$$I'_a = I_a \cup \bigcup_{a \succcurlyeq d, d\in D} O_d \quad \forall a \in B.$$

That is, we add the output of all less informed deterministic processes to the input of a black-box process.

Lemma 6.2 *If* $\mathbf{A}$ *is ordered then feedback*$(\mathbf{A})$ *is hierarchical.*

Proof: For two comparable black-box processes $b \preceq b'$, $\bigcup_{b\succcurlyeq d, d\in D} O_d \subseteq \bigcup_{b\succcurlyeq d, d\in D} O_d$ holds by construction. It thus suffices to show that $I_b \subseteq I'_{b'}$.

Let us assume that $I_b \not\subseteq I'_{b'}$, and let $p \in \Pi$ be an atomic proposition in $I_b \setminus I'_{b'}$. We distinguish two cases:

- p is in the output of a nondeterministic process ($p \in O_a, a \notin D$).

  This implies $b' \not\succcurlyeq b$. $\lightning$

- p is in the output of a deterministic process ($p \in O_d, d \in D$).

  This implies that b is not better informed than d (because $p \notin \bigcup_{b\succcurlyeq d, d\in D} O_d$), which in turn implies $b \notin U_{b'}$. $\lightning$. □

An ordered architecture $\mathbf{A}$ is equivalent to *feedback*$(\mathbf{A})$ in the sense that the same specifications are realizable for the same for set of implementations for the white-box processes.

Theorem 6.3 *Let the architecture* $\mathbf{A}$ *be ordered, and let* $\mathbf{A}' =$ *feedback*$(\mathbf{A})$. *Then an ATμC formula* φ *is realizable in* $\mathbf{A}$ *for a given set* $P_W = \{p_w\}_{w\in W}$ *of implementations for the white-box processes if and only if* φ *is realizable in* $\mathbf{A}$ *for* P_W.

Proof: The 'if' direction is trivial: It suffices that every black-box process ignores the additional information. A suitable implementation $P_B = \{p_b\}_{b \in B}$ for $\mathbf{A}$ can simply be translated to the implementation $P'_B = \{p_b \circ hide_{I'_b \setminus I_b}\}_{b \in B}$.

For the 'only if' direction, we show by induction that each black-box processes b in $\mathbf{A}$ can simulate the behavior of the black-box processes in $\mathbf{A}'$ by showing that they can infer the input sequence $I'_0 I'_1 I'_2 \dots I'_n$ in $\mathbf{A}'$ from the input sequence $hide_{I'_b \setminus I_b}(I'_0 I'_1 I'_2 \dots I'_{n-1})$. Let $P' = P_W \cup P'_B$ for a fixed $P'_B = \{p_b : (2^{I_b})^* \to 2^{O_b}\}_{b \in B}$ be an implementation for $\mathbf{A}'$.

For the induction basis, the process b in $\mathbf{A}$ can infer the empty input sequence ε that it had seen in $\mathbf{A}'$.

For the induction step, the black-box process b in $\mathbf{A}$ can infer the input sequence $I'_0 I'_1 I'_2 \dots I'_{n-1} \in (2^{I'_b})^n$ from the input sequence $hide_{I'_b \setminus I_b}(I'_0 I'_1 I'_2 \dots I'_{n-1})$ by induction hypothesis. For all deterministic processes d that are less informed than b ($b \succcurlyeq d$), b can therefore infer the reaction $O_n = p_d(hide_{I'_b \setminus I'_d} I'_0 I'_1 I'_2 \dots I'_{n-1})$ of d. Thus, b can infer $I'_0 I'_1 I'_2 \dots I'_n \in (2^{I'_b})^{(n+1)}$ from the input sequence $hide_{I'_b \setminus I_b}(I'_0 I'_1 I'_2 \dots I'_n)$. □

6.5 The Synthesis Algorithm

In this section, we present a synthesis algorithm for hierarchical architectures. The construction is based on automata over infinite trees and concurrent game structures (cf. Chapter 4).

6.5.1 Realizability in 1-Black-Box Architectures

We first consider the realizability problem for architectures with a single black-box process. Given such an architecture $\mathbf{A} = (A, \{b\}, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$, an AT$\mu$C specification φ and a set $P_W = \{p_w\}_{w \in W}$ of implementations for the white-box processes, the following algorithm constructs a nondeterministic automaton $\mathcal{F}$, which accepts an implementation p_b of the black-box process b if and only if the distributed implementation $P = P_W \cup \{p_b\}$ defines a computation tree $\langle \|p_A\|, dir \rangle$ that is a model of φ. Realizability can then be checked by solving the emptiness game for $\mathcal{F}$. The synthesis algorithm uses the following automata operations:

- **From specification to automata.** First, a specification φ is turned into an ε-free ACG $\mathcal{A}$ that accepts exactly the models of φ (Theorem 4.1 and Lemma 4.2).

- **From models to implementations.** We then transform $\mathcal{A}$ into an alternating tree automaton $\mathcal{B}$ that accepts a relaxed implementation $\langle (2^\Pi)^*, l \times \bigoplus_{a \in A} p'_a \rangle$ (extended by atomic propositions; $l : (2^\Pi)^* \to 2^\Pi$) with input Π if and only if $\langle \|p'_A\|, l \rangle$ is a model of φ (Lemma 6.4).

- **Adjusting for white-box processes.** In a third step, we construct an alternating automaton $\mathcal{C}$ that accepts a $2^\Pi \times \mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, l \times p_b \rangle$ if and only if the $2^\Pi \times \mathcal{O}_A$-labeled 2^Π-tree $\langle (2^\Pi)^*, l \times \bigoplus_{a \in A} p_a \rangle$ obtained by composing p_b with the implementations $P_W = \{p_w\}_{w \in W}$ of the white-box processes is accepted by $\mathcal{B}$ (Lemma 6.5).

- **Pruning the directions from the label.** We then construct an alternating automaton $\mathcal{D}$ that accepts an $\mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, p_b \rangle$ if and only if the $2^\Pi \times \mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, \mathit{dir} \times p_b \rangle$ obtained by adding the directions to the label is accepted by $\mathcal{C}$ (Lemma 6.6).

- **Incomplete information.** In a fifth step, we transform $\mathcal{D}$ into an alternating automaton $\mathcal{E}$ that accepts an $\mathcal{O}_b$-labeled 2^{I_b}-tree $\langle (2^{I_b})^*, p_b \rangle$ if and only if its suitable widening $\langle (2^\Pi)^*, p_b \circ \mathit{hide}_{2^{\Pi \setminus I_b}} \rangle$ is accepted by $\mathcal{D}$ (Lemma 6.7).

- **Emptiness test.** In the last step, we test the emptiness of $\mathcal{E}$ by first constructing a nondeterministic tree automaton $\mathcal{F}$ with $\mathcal{L}(\mathcal{F}) = \mathcal{L}(\mathcal{E})$ (Corollary 4.6), and then performing a constructive non-emptiness test for $\mathcal{E}$ (Theorem 4.7).

From Specifications to Automata. By Theorem 4.1 and Lemma 4.2, an ATμC formulas of length l and alternation depth d can be transformed to an equivalent ε-free automaton over concurrent game structures $\mathcal{A}$ with $O(l \cdot d)$ atoms and $O(d)$ colors (cf. Chapter 4).

From Models to Implementations. To transformation $\mathcal{A}$ into an alternating tree automaton that accepts a relaxed implementation if and only if it defines a model of φ, we construct an automaton that represents the second phase of the acceptance game for ACGs by an $\bigwedge\bigvee$ or $\bigvee\bigwedge$ representation for the choice-sets defined by the implementations (Lemma 6.4).

Lemma 6.4 *For an ε-free ACG $\mathcal{A} = (2^\Pi, Q, q_0, \delta, \alpha)$ and an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ we can construct an alternating automaton $\mathcal{B} = (2^\Pi \times \mathcal{O}_A, 2^\Pi, Q, q_0, \delta', \alpha)$ that accepts a tree $\langle (2^\Pi)^*, l \times \bigoplus_{a \in A} p_a \rangle$ if and only if*

the concurrent game tree $\mathcal{G} = (A, \Pi, (2^\Pi)^*, \varepsilon, l, \{p_a\}_{a \in A}, \tau)$ *with* $\tau : (s, d) \mapsto s \cdot d$ *is accepted by* $\mathcal{A}$.

Proof: Since the potential decisions of the processes are determined by the (relaxed) implementation, the universal and existential atoms can be resolved by Boolean combinations of concrete directions.

We obtain $\delta'(q, (V, \bigoplus_{a \in A} \mathbf{O}_a))$ by resolving the $\forall\exists$ and $\exists\forall$ semantics of universal and existential atoms in $\delta(q, \pi)$ in the following way:

- Each occurrence of $(q', \Box, A')$ in $\delta(q, \pi)$ is replaced by $\bigvee_{O_{A'} \in \bigoplus_{a \in A'} \mathbf{O}_a} \bigwedge_{O_{A \smallsetminus A'} \in \bigoplus_{a \in A \smallsetminus A'} \mathbf{O}_a} (q', O_{A'} \cup O_{A \smallsetminus A'})$.

  The outer disjunction refers to the fact that the agents in A' first choose a direction in accordance with the enabled directions in the current state. The inner conjunction refers to the counter choice made by the agents in $A \smallsetminus A'$.

- Likewise, each occurrence of $(q', \Diamond, A')$ in $\delta(q, \pi)$ is replaced by $\bigwedge_{O_{A \smallsetminus A'} \in \bigoplus_{a \in A \smallsetminus A'} \mathbf{O}_a} \bigvee_{O_{A'} \in \bigoplus_{a \in A'} \mathbf{O}_a} (q', O_{A'} \cup O_{A \smallsetminus A'})$. □

Adjusting for White-Box Processes. The $\mathcal{O}_W$-fraction of the label represents the decisions made by the white-box processes. Consequently, we are only interested in those trees, where the label of every node is in accordance with these decisions. This information is then redundant and can be pruned. We assume that the composed strategy $\bigoplus_{w \in W} p_w$ of the white-box processes is represented as a deterministic finite-state Moore machine $\mathcal{M} = (2^\Pi, O, o_0, d_W, o_W)$, where O is a set of states, o_0 the initial state, the transition function $d_W : 2^\Pi \times O \to O$ is a mapping from the input alphabet and the set of states to the set of states, and the output function $o_W : O \to \mathcal{O}_W$ maps each state to a nonempty set of output letters. The following operation performs the pruning; the state space of the resulting automaton is bilinear in the state space of the original automaton and the number of states of $\mathcal{M}$, while the set of colors remains unchanged.

Lemma 6.5 *Given an alternating automaton* $\mathcal{B} = (\Sigma \times \Xi, \Upsilon, Q, q_0, \delta, \alpha)$ *and a deterministic finite-state Moore machine* $\mathcal{M} = (\Sigma, O, o_0, d_W, o_W)$ *that produces a* Ξ*-labeled* Υ*-tree* $\langle \Upsilon^*, l \rangle$*, we can construct an alternating automaton* $\mathcal{C} = (\Sigma, \Upsilon, Q \times O, (q_0, o_0), \delta', \alpha')$ *over* Σ*-labeled* Υ*-trees, such that* $\mathcal{C}$ *accepts*

$\langle \Upsilon^*, l' \rangle$ if and only if $\mathcal{B}$ accepts $\langle \Upsilon^*, l'' \rangle$ with $l'' : y \mapsto (l'(y), l(y))$.
If $\mathcal{B}$ is a nondeterministic automaton, so is $\mathcal{C}$.

Proof: If $\delta : (q; \sigma, \xi) \mapsto b_{(q,\sigma,\xi)}(\{q_i, \upsilon_i \mid i \in I\})$, we can set $\delta' : (q, o; \sigma) \mapsto b_{(q,\sigma,o_W(o))}\big(\{(q_i, d_W(\sigma, o)), \upsilon_i \mid i \in I\}\big)$. The coloring function can simply be set to $\alpha' : (q, o) \mapsto \alpha(q)$. □

Pruning the Directions from the Label. The label of each state of the game structures is determined by the last common decision made by the processes. Recall that $\langle \Upsilon^*, dir \rangle$ denotes the Υ-labeled Υ-tree with $dir : y \cdot \upsilon = \upsilon$ for all $y \in \Upsilon^*$ and $\upsilon \in \Upsilon$, and $dir : \varepsilon \mapsto \upsilon_0$ for some predefined root direction $\upsilon_0 \in \Upsilon$.

Lemma 6.6 *[KV99] Given an alternating automaton $\mathcal{C} = (\Upsilon \times \Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ over $\Upsilon \times \Sigma$-labeled Υ-trees, we can construct an alternating automaton $\mathcal{D} = (\Sigma, \Upsilon, Q \times \Upsilon, (q_0, \upsilon_0), \delta', \alpha')$ over Σ-labeled Υ-trees such that $\mathcal{D}$ accepts a tree $\langle \Upsilon^*, l \rangle$ if and only if $\mathcal{C}$ accepts $\langle \Upsilon^*, dir \times l \rangle$.* □

Constructing $\mathcal{D}$ is simple: It suffices to store the last direction υ in the extended state $(q, \upsilon) \in Q \times \Upsilon$, and to use the direction stored in the state as substitution for the eliminated direction. That is, for $\delta\big(q, (\upsilon, \sigma)\big) = b\big((q_1, \upsilon_1), \ldots, (q_n, \upsilon_n)\big)$, we get $\delta'\big((q, \upsilon), \sigma\big) = b\big(((q_1, \upsilon_1), \upsilon_1), \ldots, ((q_n, \upsilon_n), \upsilon_n)\big)$.

If we want to allow for a set $\Upsilon_0 \subseteq \Upsilon$ of different root directions, we can add a fresh initial state q_0' and set $\delta'(q_0', \sigma) = \bigvee_{\upsilon \in \Upsilon_0} = \delta'\big((q_0, \upsilon), \sigma\big)$.

Incomplete Information. The output of the black-box process b may only depend on the input I_b visible to b. We therefore construct an automaton that accepts an implementation if and only if its widening is accepted by $\mathcal{C}$.

Lemma 6.7 *[KV99] Given an alternating automaton $\mathcal{D} = (\Sigma, \Xi \times \Upsilon, Q, q_0, \delta, \alpha)$ over Σ-labeled $\Xi \times \Upsilon$-trees, we can construct an alternating automaton $\mathcal{E} = (\Sigma, \Xi, Q, q_0, \delta', \alpha)$ over Σ-labeled Ξ-trees, such that $\mathcal{E}$ accepts $\langle \Xi^*, l \rangle$ if and only if $\mathcal{D}$ accepts $\langle (\Xi \times \Upsilon)^*, wide_\Upsilon(l) \rangle$.* □

This automata transformation changes the transition function δ. The new transition function δ' is constructed by replacing each occurrence of $(q, (\xi, \upsilon))$ in the mapping of δ by (q, ξ). A memoryless winning strategy of $\mathcal{E}$ for a Σ-labeled Ξ-tree $\langle \Xi^*, l \rangle$ defines a memoryless winning strategy for $\mathcal{D}$ on its Υ-widening

$\langle(\Xi\times\Upsilon)^*, \mathit{wide}_\Upsilon(l)\rangle$: If the winning strategy of $\mathcal{E}$ maps a node $x \in \Xi^*$ and a state $q \in Q$ to a set $\{(q_i, \xi_i) \mid i \in I\}$ of atoms, then the winning strategy of $\mathcal{D}$ maps a state $y \in (\Xi \times \Upsilon)^*$ with $\mathit{hide}_\Upsilon(y) = x$ and q to $\{(q_i, (\xi_i, \upsilon)) \mid i \in I, \upsilon \in \Upsilon\}$.

Vice versa, an accepting run-tree of $\mathcal{D}$ for $\langle(\Xi\times\Upsilon)^*, \mathit{wide}_\Upsilon(l)\rangle$ can be turned into an accepting run tree of $\mathcal{E}$ for a Σ-labeled Ξ-tree $\langle\Xi^*, l\rangle$ by replacing every node $y \in (\Xi \times \Upsilon)^*$ from each label of the run tree by $\mathit{hide}_\Upsilon(y)$.

Emptiness Test. The resulting alternating automaton $\mathcal{E}$ can be transformed into an equivalent nondeterministic automaton $\mathcal{F}$ by Corollary 4.6, for which we can perform a constructive non-emptiness test by Theorem 4.7.

6.5.2 Realizability in Hierarchical Architectures

For a hierarchical architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$, the linear informedness preorder $\succeq = \{(b, b') \in B \times B \mid I_b \supseteq I_{b'}\}$ partitions the black-box processes B into equivalence classes and defines an order on them. If $\succeq$ defines n different equivalence classes, we say that $\mathbf{A}$ has n levels of informedness. We define an ordering function $o : \mathbb{N}_n \to 2^B$, which maps each natural number $i \in \mathbb{N}_n = \{1, \ldots, n\}$ to the set of i-th best informed black-box processes. For convenience, we use $\mathcal{O}_i = \bigoplus_{b\in o(\{i,\ldots,n\})} \mathcal{O}_b$ and $I_i = I_b$ for $b \in o(i)$.

The Algorithm. We start by applying the transformations discussed in the previous subsection (Theorem 4.1, Lemma 4.2, and Lemmata 6.4 through 6.5) to construct a tree automaton $\mathcal{D}_1$ that accepts a set of relaxed implementations $P_0 = \{p_b\}_{b\in B}$ (with input Π) if and only if $P = P_W \cup P_0$ satisfies φ.

Then, we stepwise eliminate the processes in decreasing order of informedness. We successively construct:

- The alternating automaton $\mathcal{E}_i$ that accepts an $\mathcal{O}_i$-labeled 2^{I_i}-tree if and only if its widening is accepted by $\mathcal{D}_i$ (Lemma 6.7).

  A set $P_i = \{p_b^i \mid b \in B_i\}$ of relaxed implementations with input I_i for the processes in $B_i = o(\{i, \ldots, n\})$ is accepted by $\mathcal{E}_i$ if and only if there is a set $\overline{P}_i = \{\overline{p}_b^i \mid b \in \overline{B}_i\}$ of implementations for the processes in $\overline{B}_i = o(\mathbb{N}_{i-1})$, such that $P_W \cup P_i \cup \overline{P}_i$ satisfies φ.

- The nondeterministic automaton $\mathcal{F}_i$ with $\mathcal{L}(\mathcal{F}_i) = \mathcal{L}(\mathcal{E}_i)$ (Corollary 4.6); and

- The nondeterministic automaton $\mathcal{D}_{i+1}$ that accepts an $\mathcal{O}_{i+1}$-labeled I_i-tree if and only if it can be extended to an $\mathcal{O}_i$-labeled I_i-tree accepted by $\mathcal{D}_i$ (Lemma 6.8).

Narrowing and nondeterminization have been discussed in the previous section, and language projection is a standard operation on nondeterministic automata.

Lemma 6.8 *Given a nondeterministic automaton $\mathcal{F} = (\Sigma \times \Xi, \Upsilon, Q, q_0, \delta, \alpha)$ that runs on $\Sigma \times \Xi$-labeled Υ-trees, we can construct a nondeterministic automaton $\mathcal{D} = (\Sigma, \Upsilon, Q, q_0, \delta', \alpha)$ that accepts a Σ-labeled Υ-tree $\langle \Upsilon^*, l_\Sigma \rangle$ if and only if there is a $\Sigma \times \Xi$-labeled Υ-tree $\langle \Upsilon^*, l_\Sigma \times l_\Xi \rangle$ that is accepted by $\mathcal{F}$.*

Proof: $\mathcal{D}$ can be constructed by using δ' to guess the correct tree: We set $\delta' : (q, \sigma) \mapsto \bigvee_{\xi \in \Xi} \delta(q, (\sigma, \xi))$.

If $\mathcal{F}$ accepts a $\Sigma \times \Xi$-labeled Υ-tree $\langle \Upsilon^*, l \rangle$, than $\mathcal{F}$ has a memoryless winning strategy for $\langle \Upsilon^*, l \rangle$. From this strategy, we can infer a memoryless winning strategy for $\mathcal{D}$ on the Σ-projection: $\mathcal{D}$ can simply choose, for a position $y \in \Upsilon^*$ in the tree and a state q of the automaton, the same set of atoms $\mathcal{F}$ chooses.

Vice versa, if $\mathcal{D}$ accepts a Σ-labeled Υ-tree $\langle \Upsilon^*, l' \rangle$, than it has a memoryless winning strategy for it. This strategy defines a run tree, which maps each node of $\langle \Upsilon^*, l' \rangle$ to a (unique) state of the automaton. Knowing the memoryless strategy, the position y in the tree and the state q of the automaton, we can infer first the set of atoms chosen by the strategy, and then an element $\xi \in \Xi$ such that this set of atoms satisfies $\delta(q, l'(y), \xi)$. If we use this method to construct a Ξ-labeled Υ-tree $\langle \Upsilon^*, l'' \rangle$, then the $\Sigma \times \Xi$-labeled Υ-tree $\langle \Upsilon^*, l' \times l'' \rangle$ is accepted by $\mathcal{F}$ (with the memoryless winning strategy of $\mathcal{D}$ for $\langle \Upsilon^*, l' \rangle$). □

We check realizability by solving the emptiness game for $\mathcal{F}_n$. This step can be extended to the synthesis of implementations $\{p_b\}_{b \in B}$ of the black-box processes.

6.5.3 Synthesis

The specification is realizable if and only if player *accept* has a winning strategy in the emptiness game of $\mathcal{F}_n$. From this strategy we obtain by projection a family of implementations $P_n = \{p_a \mid a \in o(n)\}$ for the least-informed processes.

In increasing order of informedness, we obtain implementations for the other processes: After computing implementations for the processes in $o(\{i+1, \ldots, n\})$, they are represented as deterministic finite-state Moore machines. Using Lemma 6.5, we then construct from $\mathcal{F}_i$ a nondeterministic automaton $\mathcal{G}_i$ that accepts those implementations $\widehat{P}_i$ for the processes in $o(i)$ for

which there exists a set of implementations $\overline{P}_{i-1} = \{p_a \mid a \in o(\mathbb{N}_{i-1})\}$ such that $P_W \cup \overline{P}_{i-1} \cup \widehat{P}_i \cup P_{i+1}$ satisfies φ. $\mathcal{G}_i$ is non-empty by construction. From the winning strategy for player *accept* we obtain by projection a family of implementations $P' = \{p_a \mid a \in o(i)\}$, and set P_i to $P' \cup P_{i+1}$.

Theorem 6.9 *The distributed synthesis problem for a hierarchical architecture* **A** *with n levels of informedness, a specification φ given as an ATμC formula, and a family $P_W = \{p_w\}_{w \in W}$ of implementations of the white-box processes can be solved in time n-exponential in the number of subformulas of φ.*

Proof: The specification φ is realizable for an architecture **A** and a given set $\{p_w\}_{w \in W}$ of white-box strategies if and only if $\mathcal{F}_n$ is not empty. The construction of $\mathcal{F}_n$ involves one transformation of an alternating automaton to a nondeterministic automaton for each $i \in \mathbb{N}_n$, and therefore takes time n-exponential in the number of subformulas of φ. The size of each nondeterministic automaton $\mathcal{G}_i$ is linear in the size of $\mathcal{F}_i$ and the size of the Moore machines for the strategy of the less-informed processes. Each step along the order of informedness therefore again takes n-exponential time. □

Remark. The construction is also n-exponential in the size of the Moore machine $\mathcal{M}$ that represents the white-box strategies. This complexity can be improved by pruning the directions successively and removing the strategies of the white-box processes as late as possible, that is, directly before the first elimination of information that is available to the white-box process.

Upper bounds for ATL, CTL* and ATL* follow from linear translations to alternation-free ATμC [AHK02], exponential translations to the μ-calculus [BC96], and doubly exponential translations to ATμC [dAHM01, AHK02], respectively. μ-calculus and CTL form a syntactical subset of ATμC and ATL, respectively.

Corollary 6.10 *The distributed synthesis problem for a hierarchical architecture* **A** *with n levels of informedness and a specification φ can be performed in time n-exponential in the length of φ for specifications in CTL, ATL, or the classic μ-calculus, $(n+1)$-exponential in the length of φ for specifications in CTL*, and $(n+2)$-exponential in the length of φ for specifications in ATL*.* □

A matching nonelementary lower bound (for LTL formulas and pipelines) is provided in [PR90].

From Implementations for *feedback*(A) to Implementations for A. We start with composing the deterministic finite-state Moore machines that represent the implementations $\{p_a\}_{a \in A}$ of the individual processes to a single deterministic finite-state Moore machine $\mathcal{M}$. For $\mathcal{M} = (2^\Pi, O, o_0, d_A, o_A, l_A)$,

- O is a set of states – the product state from the individual implementations,
- the initial state o_0 is the tuple of individual initial states,
- the output function $o_A : O \to \mathcal{O}_A$ maps each state to a nonempty set of possible decision of the processes, and is composed from the individual output functions,
- the transition function $d_A : 2^\Pi \times O \rightharpoonup O$ is a partial mapping from the input alphabet and the set of states to the set of states, such that $(\pi, o) \in 2^\Pi \times O$ is in the preimage of the (partial) transition function d_A if and only if $\pi \in o_A(o)$ is a possible decision in o, and
- $l_A : O \to 2^\Pi$ is a labeling function that maps each state to the set of atomic propositions valid in it.

By construction of $\mathcal{M}$, the size $|O|$ of $\mathcal{M}$ is n-exponential in the length of an ATμC specification φ, and $\mathcal{M}$ is input preserving, that is, $l\big(d_A(\pi, o)\big) = \pi \, \forall o \in O$, $\pi \in o_A(o)$, and $l(o_0) = \pi_0$.

Theorem 6.11 *The distributed synthesis problem for a fork-free architecture* **A** *whose (hierarchical) architecture feedback*(**A**) *has n levels of informedness and a specification φ can be performed in time n-exponential in the length of φ for specifications in CTL, ATL, or the classic or alternating-time μ-calculus, $(n+1)$-exponential in the length of φ for specifications in CTL*, and $(n+2)$-exponential in the length of φ for specifications in ATL*.*

Proof: By Corollary 6.10, the representations of the implementations of the black-box processes – and thus $\mathcal{M}$ – have the claimed size. We assume without loss of generality that all states of $\mathcal{M}$ are reachable from the initial state. (Otherwise, we can restrict $\mathcal{M}$ to its reachable fragment.)

By Theorem 6.3, the O_b part of the label and the 2^{O_b} part of the output of $\mathcal{M}$ does, for each black-box process b, only depend on the visible fragment $hide_{2^\Pi \setminus I_b}$ of the input history. Consequently, we can compute, for each black-box process b, the coarsest bisimulation relation $\sim_b$ that distinguishes two states $o_1, o_2 \in O$ of $\mathcal{M}$ if they

- differ in the O_b part of the label – $l(o_1) \cap O_b \neq l(o_2) \cap O_b$,
- differ in the 2^{O_b} part of the set of possible decisions – $o_A(o_1) \cap 2^{O_b} \neq o_A(o_2) \cap 2^{O_b}$, or
- have successors that are not identified by $\sim_b$ in directions indistinguishable by I_b – $\exists \pi_1 \in o_A(o_1), \pi_2 \in o_A(o_2). \pi_1 \cap I_b = \pi_2 \cap I_b \wedge d_A(\pi_1, o_1) \nsim_b d_A(\pi_1, o_1)$.

Computing the coarsest bisimulation relation is cheap [PT87], and a finite state implementation for $p_b : (2^{I_b})^* \to \mathcal{O}_b$ with states $O/\sim_b$ can be read from $\mathcal{M}$ and $\sim_b$. □

6.6 Completeness

The algorithm from Section 6.4 solves the synthesis problem for all architectures without information forks. In this section, we show that the occurrence of an information fork is a sufficient condition for the undecidability of an architecture, and hence establish the completeness of our approach.

Pnueli and Rosner [PR90] showed undecidability for the architecture $\mathbf{A}_0$ of Figure 6.1 and LTL, using a reduction from the halting problem. In the proof of Lemma 6.12 we give a new reduction that applies to both CTL and LTL. Lemma 6.13 and Theorem 6.14 extend this result to all architectures that contain an information fork.

Older works focus on deterministic processes. That is, the only nondeterministic process allowed for is traditionally an environment *env* as a special process that always enables all decisions. (In our setting, the environment *env* can be viewed as a white box process, whose implementation p_{env} is a constant function that maps every input history to $2^{O_{env}}$.) To cover this classical setting, the completeness proof is therefore divided into two parts:

- In Subsection 6.6.1, we show that even in this restricted setting (environment realizability), architectures that contain an information fork are undecidable for LTL and CTL specifications.
- In Subsection 6.6.2, we demonstrate completeness for general architectures that may contain nondeterministic black-box processes.

6.6.1 Environment Undecidable Architectures

The *environment realizability problem* is to decide for the triple $(\mathbf{A}, \varphi, \{p_w\}_{w \in W})$, consisting of an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ that has only deterministic black-box processes ($B \subseteq D$), a specification φ, and a set of white-box implementations $P_W = \{p_w\}_{w \in W}$, where all implementations p_d of nondeterministic white-box processes $d \in W \setminus D$ are constant functions that map every input history to 2^{O_d}, whether there exists a set $P_B = \{p_b\}_{b \in B}$ of implementations for the black-box process in $\mathbf{A}$, such that the joint behavior $\langle \|p_A\|, \mathit{dir} \rangle$ of the system satisfies φ. (In this case, all nondeterministic white-box processes can be merged into a single environment process.)

An architecture $\mathbf{A}$ that has only deterministic black-box processes is called *environment decidable* if realizability can be decided for all formulas φ from a given specification language and suitable implementations P_W of the white-box processes.

Lemma 6.12 *The architecture $\mathbf{A}_0$ is not environment decidable for CTL and LTL specifications.*

Proof: For a given deterministic Turing machine M, we define a specification φ_M that is realizable if and only if M halts on the empty input tape.

In the architecture $\mathbf{A}_0$ (see Figure 6.1), the environment p_1 communicates independently with two system processes p_2 and p_3 through their input variables a and b, respectively. In a first step, we define a specification ψ_M that has exactly one (not necessarily finite-state) implementation in which the environment p_1 can prompt the processes p_2 and p_3 to output the entire computation of M (that is, a series of successive configurations) on the (hidden) variables c and d, respectively, by sending a *start* command through the input variables a and b, respectively. Further *start* commands have no effect.

A configuration C is output as follows: It starts with the (possibly empty) sequence of tape symbols left of the read/write head, followed by first the internal state of M, and then the sequence of tape symbols from the position of the read/write head up to the first *blank*-sign.

Let $\perp$ denote the terminal state of the Turing machine and let $C \vdash C'$ denote that C' is the configuration succeeding C.

The specification $\psi_M = \psi_{p_2} \wedge \psi_{p_3}$ is constructed as the conjunction of the assertions ψ_{p_2} and ψ_{p_3}, where ψ_{p_2} is defined as follows:

- Initially, p_2 outputs $\perp$ symbols, until the first *start* symbol is received. Then, p_2 outputs the initial configuration of M and the second configuration of M, followed by a sequence of legal configurations of M.

- If p_2 and p_3 output C and C' respectively (starting concurrently) and $C \vdash C'$ holds, then the configurations C_{new} and C'_{new}, output next by p_2 and p_3, respectively, have to satisfy $C_{new} \vdash C'_{new}$.

  Note that their output starts concurrently if and only if the head was not at the end of the tape (that is, over the *blank*) in C', and C'_{new} is output with a delay of exactly one symbol otherwise.

ψ_{p_3} is the corresponding assertion, where the roles of p_2 and p_3 are swapped.

A simple inductive argument shows that ψ_M has only the canonical implementation, where both processes output the computation of M:

Assume there is an implementation, where both processes always output the first i configurations following the canonical implementation, but one process (without loss of generality p_2) fails to output the $(i+1)$-th configuration. If p_1 sends the *start* command on b exactly n steps after sending the *start* command on a, where n is the number of steps needed to output the $(i-1)$-th configuration, then p_3 writes the $(i-1)$-th configuration at the same time as p_2 outputs the i-th configuration. Hence, p_2 is forced to output the $(i+1)$-th configuration correctly as well.

Consequently, the specification φ_M, ensuring that

- ψ_M holds and

- p_2 and p_3 always eventually output $\perp$,

has a (finite state) implementation if and only if M halts on the empty input tape. The specification φ_M can easily be expressed in both CTL and LTL. □

The argument that the canonical implementation is the only possible implementation relies on the fact that p_2 is oblivious of b and d, and p_3 is oblivious of a and c. In the architecture $\mathbf{A}_0$, a and b are hidden because the environment communicates with both processes separately, and c and d are hidden because neither process is aware of the output of the other process. We generalize the argument to all architectures with an information fork by first showing that the architecture $\mathbf{A}_0$ remains environment undecidable if the output becomes visible and, in a second step, by allowing for indirect communication between the environment and the two processes.

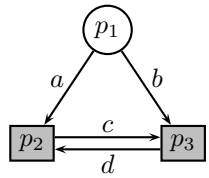


Figure 6.6: Architecture $\mathbf{A}_1$

Lemma 6.13 *The architecture* $\mathbf{A}_1 = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ *with* $A = \{p_1, p_2, p_3\}$, $B = D = \{p_2, p_3\}$, $\Pi = \{a, b, c, d\}$, $I_{p_1} = \emptyset$[3], $I_{p_2} = \{a, d\}$, $I_{p_3} = \{b, c\}$, $O_{p_1} = \{a, b\}$, $O_{p_2} = \{c\}$, *and* $O_{p_3} = \{d\}$ *of Figure 6.6 (architecture architecture* $\mathbf{A}_0$ *plus communication between* p_2 *and* p_3*) is not environment decidable for CTL and LTL specifications.*

Proof: We interpret the input a and b, respectively, in odd and even cycles in different ways: While only the input in odd cycles may be read as the *start* signal that triggers the output of sequences of configurations, the input in even cycles is interpreted as an XOR key for the following output.

Using only every second bit of the output to encode the configuration, the nondeterminism is exploited to provide us with a perfect encryption. In this setting, we can state the specification as in the proof of Lemma 6.12, with the difference that the *decrypted* version of the output has to satisfy the output-requirements. Even though the processes may read each others *encrypted* output, they are oblivious of its *decrypted* meaning. □

For the undecidability of an architecture it already suffices if the environment can pass separate information to two different processes. This extends the class of environment undecidable architectures further to those containing an information fork.

Theorem 6.14 *The distributed environment realizability problem for LTL and for CTL specifications is undecidable for all architectures* $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ *with* $B \subseteq D$ *that contain an information fork.*

Proof: If (A', p, p', q, q', E) is an information fork, we can fix two shortest paths (that may have length 0) $p_0^q p_1^q p_2^q \dots p_m^q, p_0^{q'} p_1^{q'} p_2^{q'} \dots p_n^{q'} \in (A' \setminus D)(A' \cap D)^*$, with $(p_i^q, p_{i+1}^q) \in E$ for all $i < m$, $(p_i^{q'}, p_{i+1}^{q'}) \in E$ for all $i < n$, $p_m^q = q$, and $p_n^{q'} = q'$.

[3]The input I_{p_1} of the environment p_1 is not important, because its output is constant.

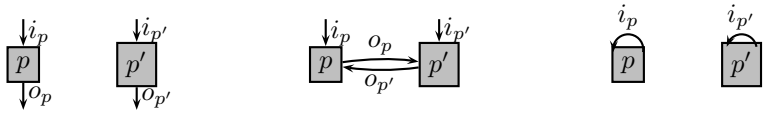


Figure 6.7: Three undecidable situations: An architecture is undecidable if it contains two processes with incomparable sets of inputs.

We use these paths to transfer secrets represented by the valuation of two (not necessarily different) environment variables $a \in (O_{p_0^q} \cap I_{p_1^q}) \setminus (I_p \cup I_{p'})$ and $b \in (O_{p_0^{q'}} \cap I_{p_1^{q'}}) \setminus (I_p \cup I_{p'})$ via $p_0^q p_1^q p_2^q \dots p_m^q$ to p and via $p_0^{q'} p_1^{q'} p_2^{q'} \dots p_n^{q'}$ to q', respectively. (To cover the cases that the variables are not different ($a = b$) or that the paths $p_0^q p_1^q p_2^q \dots p_m^q$ and $p_0^{q'} p_1^{q'} p_2^{q'} \dots p_n^{q'}$ have to share transport variables, we only use the values of a in even, and the value of b in odd cycles.)

Undecidability therefore follows as in Lemma 6.13. □

6.6.2 Undecidable Architectures

As discussed in Subsection 6.2.5, we can assume for all universal specification languages that all black-box processes are deterministic. For LTL, the undecidability result of Theorem 6.14 therefore covers all architectures that contain an information fork. For the non-universal specification language CTL, we proof the completeness of the synthesis procedure introduced in Section 6.4 by showing that an architecture is decidable if and only if it is fork-free.

The proof is an extension of the reductions from the previous subsection. In addition to the architectures considered there, we have to take into account the situation where the two processes do not receive any input from an external environment (Figure 6.7c). To cover this case, we specify that the *start*-symbols and XOR keys are chosen completely nondeterministically during the first and second phase. The configurations of the Turing machine are emitted in a separate third phase, where the values of the output variables are specified to be chosen deterministically.

Theorem 6.15 *The synthesis problem for CTL specifications is undecidable for an architecture* $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ *that contains an information fork* (A', p, p', q, q', E).

Proof: We use a reduction similar to the one in Theorem 6.14: We again fix two shortest paths (that may have length 0) $p_0^q p_1^q p_2^q \dots p_m^q, p_0^{q'} p_1^{q'} p_2^{q'} \dots p_n^{q'} \in$

$(A' \setminus D)(A' \cap D)^*$, with $(p_i^q, p_{i+1}^q) \in E$ for all $i < m$, $(p_i^{q'}, p_{i+1}^{q'}) \in E$ for all $i < n$, $p_m^q = q$, and $p_n^{q'} = q'$ and use these paths to transmit a secret to p that remains hidden from p' and, vice versa, transmit a secret to p' that remains hidden from p.

In the setting with nondeterministic processes, there may arise an additional problem: p and p' can be part of A', and they can, in principle, be the source of nondeterminism used to establish a secret. To handle this problem, we implement a three-phase protocol, where the first two phases are again used to transmit secrets, while emitting the encrypted configurations needs to be moved to a separated third phase:

- In the first phase, a *start* symbol is transmitted through the unobservable pathways to p and p', respectively.
- In the second phase, an XOR key is transmitted to p and p' through i_p and $i_{p'}$.
- p and p' output an encoded letter of their output sequence in the third phase.

We extend the specification with the following guarantees:

- The values of the variables o_p and $o_{p'}$ are fixed deterministically if p and p' output an encrypted bit in the third phase.
- The values of the input variables $I_p \cup I_{p'}$ to p and p' are chosen deterministically with the possible exception of cycles, where they serve as a source of nondeterminism (for the secret).
- If the values of the input variables $I_p \cup I_{p'}$ to p and p' are chosen deterministically, then they are set to *true* with the possible exception of cycles where they are used to
  - transmit a secret to p or p', respectively, or
  - are used to emit the encrypted output bit in the third phase (if they coincide with o_p or $o_{p'}$, respectively).

If the white-box strategies are chosen accordingly, the synthesis problem has a solution if and only if M halts on the empty input tape. □

The Theorems 6.14 and 6.15 imply the completeness of the proposed synthesis algorithm:

Corollary 6.16 *The algorithm from Section 6.4 solves the synthesis problem for all decidable architectures.*

6.7 Conclusions

The invention of model checking in the 1980s has brought formal methods to industrial practice. Hardware and many communication protocols can be modeled as finite-state automata and their automatic analysis makes formal verification economically feasible. A major drawback of model checking methods is that they require the *complete* design to be known before they can be applied. It is, however, crucial to find design errors early, before much effort has gone into the implementation.

The results from this chapter make *incomplete* designs accessible to automated analysis. As soon as enough components have been implemented to make the architecture decidable, we can automatically complete the design by deriving an implementation for the remaining processes. If synthesis fails, the unrealizability of the specification demonstrates an error in the existing partial design.

Will it be possible to completely automatize the construction of distributed systems? The results of this chapter mark the limits of system synthesis, because the introduced algorithm is already applicable to all decidable architectures. Automated program construction is still likely to work in many practical applications. An example is the system maintenance phase, which dominates the life-time cost of most systems today. Since in every maintenance cycle only a few components are modified, nearly all components remain white-box and the architecture is likely to be decidable.

The description of the set of decidable architectures can also be used to streamline the design process: We can now identify (sets of) processes that, when turned white-box, render undecidable architectures decidable. For the 5-process two-way ring of Figure 6.3f, for example, implementing one of the processes p_4 or p_5 breaks the information fork, resulting in a decidable architecture, whereas implementing the three processes p_2, p_3, and p_6 leaves an undecidable architecture. This observation can also be exploited by semi-algorithms for undecidable architectures: If a finite-state solution exists, it can be found by a simple enumeration of the process strategies. Our results show that it is not necessary to enumerate the strategies of all processes. Since enumerating the strategies of a black-box process turns that process white-box, it suffices to consider a sufficient

subset of the processes, such that all information forks are eliminated from the architecture.

Another option is the usage of incomplete methods for unrealizable architectures (or for a reduction in the number of different levels of informedness to accelerate synthesis). We can use overapproximations (where the single black-boxes receive additional information) of an architecture to demonstrate that a synthesis problem is unrealizable, and underapproximations (where the input to the black-boxes is reduced) to construct an implementation.

Chapter 7

Bounded Synthesis

Abstract

The bounded synthesis problem is to construct an implementation that satisfies a given temporal specification and a given bound on the number of states. We present a solution to the bounded synthesis problem for linear-time temporal logic (LTL), based on a novel emptiness-preserving translation from LTL to safety tree automata. For distributed architectures, where standard unbounded synthesis is in general undecidable, we show that bounded synthesis can be reduced to a SAT problem. As a result, we obtain an effective algorithm for the bounded synthesis from LTL specifications in arbitrary architectures. By iteratively increasing the bound, our construction can also be used as a semi-decision procedure for the unbounded synthesis problem. We also show that the algorithm extends to the general distributed synthesis problem with ATμC specifications introduced in the Chapter 6.

7.1 Introduction

Verification and synthesis both provide a formal guarantee that a system is implemented correctly. The difference between the two approaches is that while verification proves that a *given* implementation satisfies the specification, synthesis automatically *derives* one such implementation. Synthesis thus has the obvious advantage that it completely eliminates the need for manually writing and debugging code.

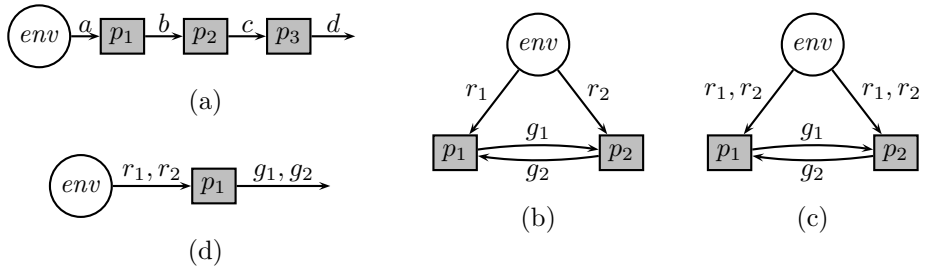


Figure 7.1: Distributed architectures: (a) pipeline architecture, (b) 2-process arbiter architecture, (c) 2-process arbiter architecture with complete information, (d) single-process architecture.

Unfortunately, the synthesis problem is undecidable even for simple distributed architectures. Consider, for example, the typical 2-process arbiter architecture shown in Figure 7.1b: The environment (*env*) sends requests (r_1, r_2) for access to a critical resource to two processes p_1 and p_2, which react by sending out grants (g_1, g_2). As shown in Lemma 6.13, the synthesis problem is undecidable for this architecture, because both p_1 and p_2 have access to information (r_1 and r_2, respectively) that is hidden from the other process. For system architectures without such *information forks* (Chapter 6), like pipeline architectures (Figure 7.1a shows a pipeline of length 3), the synthesis problem is decidable, but has nonelementary complexity.

The high complexity of synthesis is explained by the fact that, as pointed out by Rosner [Ros92], a small LTL formula of size n which refers to m different processes already suffices to specify a system that cannot be implemented with less than m-exp(n) states. From a practical point of view, however, it is questionable whether such huge implementations should be considered by the synthesis algorithm, because they are likely to violate other design considerations (such as the available memory). In this chapter, we therefore study a variation of the synthesis problem, which we call the *bounded synthesis problem*, where an upper limit on the size of the implementation is set in advance. The bound may either be an explicit design constraint or the result of iteratively increasing the limit in the search for a solution of minimal size.

Our starting point is the representation of the LTL specification as a universal Co-Büchi tree automaton. We show that the acceptance of a finite-state transition system by a universal Co-Büchi automaton can be characterized by the existence of an annotation that maps each pair of a state of the automa-

ton and a state of the transition system to a natural number. The advantage of this characterization is that the acceptance condition can be simplified to a simple safety condition: We show that the universal Co-Büchi automaton can be translated to an (emptiness-equivalent) deterministic *safety automaton* that implicitly builds a valid annotation. The emptiness of the safety automaton can then be determined in a simple two-player game, where player *accept* represents the system implementation and wins the game if the specification is satisfied; the opponent, player *reject*, wins the game if the specification is violated.

If the system architecture consists of a single process, as in Figure 7.1d, then a victory for player *accept* means that the specification is realizable. Any winning strategy for player *accept* immediately defines a correct implementation for the process. If the architecture consists of more than one process, as in the arbiter architecture of Figure 7.1b, then a victory for player *accept* only means that the specification can be implemented in the slightly modified architecture (shown for the arbiter example in Figure 7.1c), where all processes have the same information. An implementation for the architecture with incompletely informed processes must additionally satisfy a consistency requirement: If a process cannot distinguish between two different computation paths, it must react in the same way.

Inspired by the success of bounded model checking [CFG$^+$01, BCC$^+$03], we show that the bounded synthesis problem for *distributed architectures* can be effectively reduced to a SAT problem. We define a constraint system that describes the existence of a valid annotation and, additionally, ensures that the resulting implementation is consistent with the limited information available to the distributed processes. For this purpose, we introduce a mapping that decomposes the states of the safety game into the states of the individual processes: Because the reaction of a process only depends on its local state, the process is forced to give the same reaction whenever it cannot distinguish between two paths in the safety game. The satisfiability of the constraint system can be checked using standard SAT solvers [GPFW97, MMZ$^+$01]. As a result, we obtain an effective algorithm for the bounded synthesis from LTL specifications in arbitrary distributed architectures. By iteratively increasing the bound, our construction can also be used as a semi-decision procedure for the standard (unbounded) synthesis problem.

Furthermore, for architectures with a decidable synthesis problem (cf. Chapter 6), there is a computable bound for which the two problems coincide. For the unbounded synthesis problem, the algorithm thus provides both a decision procedure for fork-free architectures and a semi-decision procedure for all other architectures.

Finally, we show that it is simple to extend the method proposed in this chapter to full the full distributed synthesis problem discussed in Chapter 6.

Related work. The *synthesis of distributed reactive systems* was pioneered by Pnueli and Rosner [PR90], who showed that the synthesis problem is undecidable in general and has nonelementary complexity for pipeline architectures. An automata-based synthesis algorithm for pipeline and ring architectures is due to Kupferman and Vardi [KV01]; Walukiewicz and Mohalik provided an alternative game-based construction [WM03]. We showed in Chapter 6 that the synthesis problem is decidable if and only if the architecture does not contain an information fork. Madhusudan and Thiagarajan [MT01] consider the special case of *local* specifications (each property refers only to the variables of a single process). Among the class of acyclic architectures (without broadcast) this synthesis problem is decidable for exactly the doubly-flanked pipelines. Castellani, Mukund and Thiagarajan [CMT99] consider *transition systems* as the specification language: An implementation is correct if the product of the processes is bisimilar to the specification. In this case, the synthesis problem is decidable independently of the architecture.

Our translation of LTL formulas to tree automata is based on Kupferman and Vardi's *Safraless decision procedures* [KV05]. We use their idea of avoiding Safra's determinization using universal Co-Büchi automata. Our construction improves on [KV05] in that it produces deterministic safety automata instead of nondeterministic Büchi automata.

7.2 Preliminaries

We consider the synthesis of distributed reactive systems that are specified in linear-time temporal logic (LTL). For simplicity, we focus on *environment architectures*, where a dedicated environment process *env* is the only nondeterministic process, and the only white-box process. The environment is always maximal, that is, p_{env} maps every input history to $2^{O_{env}}$. Given an environment architecture $\mathbf{E}$ and an LTL formula φ, we decide whether there is an implementation for each system process in $\mathbf{A}$, such that the composition of the implementations satisfies φ.

7.2.1 Environment Architectures

An *environment architecture* $\mathbf{E} = (B, \Pi, \{I_b\}_{b \in B}, \{O_b\}_{b \in B \cup \{env\}})$ is an abbreviation for an architecture $\mathbf{A} = (B \uplus \{env\}, B, B, \Pi, \{I_b\}_{b \in B \cup \{env\}}, \{O_b\}_{b \in B \cup \{env\}})$ with $I_{env} = \emptyset$, and the implementation $P_W = \{p_{env}\}$ that maps every input history to $2^{O_{env}}$.

We call an environment architecture *fully informed* if every black-box process has access to the complete output of the environment ($I_b = O_{env} \, \forall b \in B$). Note that this implies that the process has access to the full system state (cf. Chapter 6).

Since every process in a fully informed architecture has enough information to simulate every other process, we can assume without loss of generality that a fully informed architecture contains only a single black-box process b.

7.2.2 Implementations as Labeled Transition Systems

Considering linear-time specifications and having a fixed set of directions, we can use ordinary transition systems instead of concurrent game structures to represent implementations.

For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-*transition system* is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a set of states T, an initial state $t_0 \in T$, a transition function $\tau : T \times \Upsilon \to T$, and a labeling function $o : T \to \Sigma$. $\mathcal{T}$ is a *finite-state* transition system if and only if T is finite.

Each system process $b \in B$ is implemented as a 2^{O_b}-labeled 2^{I_b}-transition system $\mathcal{T}_b = (T_b, t_b, \tau_b, o_b)$. The specification φ refers to the composition of the system processes, which is the 2^{Π}-labeled $2^{O_{env}}$-transition system $\mathcal{T}_E = (T, t_0, \tau, o)$, defined as follows: the set $T = \bigotimes_{b \in B} T_p \times 2^{O_{env}}$ of states is formed by the product of the states of the process transition systems and the possible values of the output variables of the environment. The initial state t_0 is formed by the initial states t_p of the process transition systems and a designated *root direction* $\subseteq O_{env}$. The transition function updates, for each system process b, the T_b part of the state in accordance with the transition function τ_b, using (the projection of) o as input, and updates the $2^{O_{env}}$ part of the state with the output of the environment process. The labeling function o labels each state with the union of its $2^{O_{env}}$ part with the labels of its T_p parts.

With respect to the system processes, the combined transition system thus simulates the behavior of all process transition systems; with respect to the environment process, it is *input-preserving*, that is, in every state, the label accurately reflects the input received from the environment.

7.2.3 Synthesis

A specification φ is (finite-state) *realizable* in an environment architecture $\mathbf{E} = (B, \Pi, \{I_b\}_{b \in B}, \{O_b\}_{b \in B \cup \{env\}})$ if and only if there exists a family of (finite-state) implementations $\{\mathcal{T}_b \mid b \in b\}$ of the system processes, such that their composition $\mathcal{T}_E$ satisfies φ.

7.2.4 Bounded Synthesis

We introduce bounds on the size of the process implementations and on the size of the composition. For a given architecture $\mathbf{E} = (B, \Pi, \{I_b\}_{b \in B}, \{O_b\}_{b \in B \cup \{env\}})$, a specification φ is *bounded realizable* with respect to a family of bounds $\{b_a \in \mathbb{N} \mid a \in B\}$ on the size of the system processes and a bound $b_E \in \mathbb{N}$ on the size of the composition T_E, if there exists a family of implementations $\{\mathcal{T}_a \mid a \in B\}$, where, for each process $a \in B$, $\mathcal{T}_a$ has at most b_a states, such that their composition $\mathcal{T}_E$ satisfies φ and has at most b_E states.

7.3 Annotated Transition Systems

In this section, we introduce an annotation function for transition systems. The annotation function has the useful property that a finite-state transition system satisfies the specification if and only if it has a valid annotation.

Our starting point is a representation of the specification as a universal alternating Co-Büchi automaton. Since the alternating automaton is universal, every transition system in the language of the automaton has a unique run tree. This run tree r is self-similar: Two nodes y, y' of the run tree that are decorated with the same label $r(y) = r(y') = (q, t)$ have sets $S(y), S(y')$ of successors that are equally labeled ($\{r(s) \mid s \in S(y)\} = \{r(s) \mid s \in S(y')\}$). We can therefore view the run tree as (the unraveling of) a run graph, with the same acceptance condition.

The annotation assigns to each pair (q, t) of a state q of the automaton and a state t of the transition system either a natural number or a blank sign. The natural number indicates the maximal number of rejecting states that occur on some path to (q, t) in the run graph.

We show that the finite-state transition systems accepted by the automaton are exactly those transition systems for which there is an annotation that assigns only natural numbers to the vertices of the run graph. We call such annotations *valid*.

Our construction has two steps. In the first step, we translate the specification to a *universal Co-Büchi automaton*. This standard construction is language-preserving: The automaton accepts exactly the transition systems that satisfy the specification.

In the second step, we translate the universal Co-Büchi automaton to a parametrized deterministic safety automaton. The relationship between the Co-Büchi automaton and the safety automaton is the following: For every parameter value, the safety automaton recognizes a sublanguage of the Co-Büchi automaton; for sufficiently high parameter values, the Co-Büchi automaton and the safety automaton become emptiness-equivalent. The advantage of this parametrized construction is that it is possible to increase the value of the parameter (and therefore also the size of the emptiness game) incrementally.

7.3.1 Recap: Universal Co-Büchi Automata

We translate a given LTL specification φ into an equivalent universal Co-Büchi automaton $\mathcal{U}_\varphi$. This can be done with a single exponential blow-up by first negating φ, then translating $\neg\varphi$ into an equivalent nondeterministic Büchi word automaton [GO01], and then dualizing the resulting nondeterministic Büchi automaton into a universal Co-Büchi automaton [MS87] (cf. Lemma 9.3) that accepts a transition system $\mathcal{T}$ if and only if the specification φ holds along every path of $\mathcal{T}$.

Theorem 7.1 *[KV05] Given an LTL formula φ, we can construct a universal Co-Büchi automaton $\mathcal{U}_\varphi$ with $2^{O(|\varphi|)}$ states that accepts a transition system $\mathcal{T}$ if and only if $\mathcal{T}$ satisfies φ.* □

7.3.2 Bounded Annotations

As a preparation for the translation of universal Co-Büchi tree automata to deterministic safety tree automata, we introduce an annotation function that assigns to each pair (q, t) of a state q of the automaton and a state t of the transition system either a natural number or a blank sign. Intuitively, the annotation indicates the maximal number of rejecting states that occur on some path from the initial state to (q, t) in the run graph.

An *annotation* of a transition system $\mathcal{T} = (T, t_0, \tau, o)$ on a universal Co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ is a function $\lambda : Q \times T \to \{\_\} \cup \mathbb{N}$. We call an annotation *c-bounded* if its mapping is contained in $\{\_\} \cup \{0, \ldots, c\}$, and

bounded if it is c-bounded for some natural number $c \in \mathbb{N}$. An annotation is *valid* if it satisfies the following conditions:

- the pair (q_0, t_0) of initial states is annotated with a natural number ($\lambda(q_0, t_0) \neq \_$), and
- if a pair (q, t) is annotated with a natural number ($\lambda(q, t) = n \neq \_$) and $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $(q', \tau(t, \upsilon))$ is annotated with a greater number, which needs to be strictly greater if $q' \in F$ is rejecting. That is, $\lambda(q', \tau(t, \upsilon)) \triangleright_{q'} n$ where $\triangleright_{q'}$ is $>$ for $q' \in F$ and $\geq$ otherwise.

Theorem 7.2 *A finite-state Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is accepted by a universal Co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ if and only if it has a valid $(|T| \cdot |F|)$-bounded annotation.*

Proof: Since $\mathcal{U}$ is universal, $\mathcal{U}$ has a unique run graph $\mathcal{G} = (G, E)$ on $\mathcal{T}$. Since $\mathcal{T}$ and $\mathcal{U}$ are finite, $\mathcal{G}$ is finite, too.

If $\mathcal{G}$ contains a lasso with a rejecting state in its loop, that is, a path $(q_0, t_0)(q_1, t_1) \dots (q_n, t_n) = (q'_0, t'_0)$ and a path $(q'_0, t'_0)(q'_1, t'_1) \dots (q'_m, t'_m) = (q'_0, t'_0)$ such that q'_i is rejecting for some $i \in \{1, \dots, m\}$, then, by induction, any valid annotation λ satisfies

- $\lambda(q_j, t_j) \in \mathbb{N}$ for all $j \in \{0, \dots, n\}$,
- $\lambda(q'_j, t'_j) \in \mathbb{N}$ for all $j \in \{0, \dots, m\}$,
- $\lambda(q'_{j-1}, t'_{j-1}) \leq \lambda(q'_j, t'_j)$ for all $j \in \{1, \dots, m\}$, and
- $\lambda(q'_{i-1}, t'_{i-1}) < \lambda(q'_i, t'_i)$. $\lightning$

If, on the other hand, $\mathcal{G}$ does not contain a lasso with a rejecting state in its loop, we can infer a valid $(|T| \cdot |F|)$-bounded annotation by assigning to each vertex $(q, t) \in G$ of the run graph the highest number of rejecting states occurring on some path $(q_0, t_0)(q_1, t_1) \dots (q, t)$, and by assigning $\_$ to every pair of states $(q, t) \notin G$ not in $\mathcal{G}$. □

7.3.3 Estimating the Bound

Since the distributed synthesis problem is undecidable, it is in general not possible to estimate a sufficient bound c that guarantees that a transition system with a valid c-bounded annotation exists if the specification is realizable.

For fully informed architectures, however, such an estimate is possible. If a universal Co-Büchi automaton is non-empty, then the size of a smallest accepted transition system can be estimated by the size of an equivalent deterministic parity automaton.

Theorem 7.3 *[Pit06] Given a universal Co-Büchi automaton $\mathcal{U}$ with n states, we can construct an equivalent deterministic parity automaton $\mathcal{P}$ with n^{2n+2} states and $2n$ colors.* □

A solution to the synthesis problem is required to be input-preserving, that is, in every state, the label must accurately reflect the input. Input preservation can be checked with a deterministic safety automaton $\mathcal{D}_\mathcal{I}$, whose states are formed by the possible inputs $\mathcal{I} = 2^{O_{env}}$. In every state $i \in \mathcal{I}$, $\mathcal{D}_\mathcal{I}$ checks if the label agrees with the input i, and sends the successor state $i' \in \mathcal{I}$ into the direction i'. If $\mathcal{U}$ accepts an input-preserving transition system, then we can construct a *finite* input-preserving transition system, which is accepted by $\mathcal{U}$, by evaluating the emptiness game of the product automaton of $\mathcal{P}$ and $\mathcal{D}_\mathcal{I}$. The minimal size of such an input-preserving transition system can be estimated by the size of $\mathcal{P}$ and $\mathcal{I}$.

Corollary 7.4 *If a universal Co-Büchi automaton $\mathcal{U}$ with n states and m rejecting states accepts an input-preserving transition system, then $\mathcal{U}$ accepts a finite input-preserving transition system $\mathcal{T}$ with $n^{2n+2} \cdot |\mathcal{I}|$ states, where $\mathcal{I} = 2^{O_{env}}$. $\mathcal{T}$ has a valid $m \cdot n^{2n+2} \cdot |\mathcal{I}|$-bounded annotation for $\mathcal{U}$.* □

7.4 Automata-Theoretic Bounded Synthesis

Using the annotation function, we can reduce the synthesis problem for fully informed architectures to a simple emptiness check on safety automata. The following theorem shows that there is a deterministic safety automaton that, for a given parameter value c, accepts a transition system if and only if it has a valid c-bounded annotation. This leads to the following automata-theoretic synthesis procedure for fully informed architectures:

Given a specification, represented as a universal Co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$, we construct a sequence of safety automata that check for valid bounded annotations up to the bound $c = |F| \cdot b$, where b is either the predefined bound b_E on the size of the transition system, or the sufficient bound $n^{2n+2} \cdot |\mathcal{I}|$ from Corollary 7.4. If the intersection of $\mathcal{D}_\mathcal{I}$ with one of these automata is non-empty, then the specification is realizable; if the intersection with

the safety automaton for the largest parameter value c is empty, then the specification is unrealizable. The emptiness of the automata can be checked by solving their emptiness games.

Theorem 7.5 *Given a universal Co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \delta, F)$, we can construct a family of deterministic safety automata $\{\mathcal{D}_c = (\Sigma, \Upsilon, S_c, s_0, \delta_c) \mid c \in \mathbb{N}\}$ such that $\mathcal{D}_c$ accepts a transition system if and only if it has a valid c-bounded annotation.*

Construction: We choose the functions from Q to the union of $\mathbb{N}$ and a blank sign ($S = Q \to \{\_\} \cup \mathbb{N}$) as the state space of an abstract deterministic safety automaton $\mathcal{D} = (\Sigma, \Upsilon, S, s_0, \delta_\infty)$. Each state of $\mathcal{D}$ indicates how many times a rejecting state may have been visited in some trace of the run graph that passes the current position in the transition system. The initial state of $\mathcal{D}$ maps the initial state of $\mathcal{U}$ to 0 ($s_0(q_0) = 0$) and all other states of $\mathcal{U}$ to blank ($\forall q \in Q \setminus \{q_0\}.\, s_0(q) = \_$).

Let $\delta^+_\infty(s, \sigma) = \{((q', s(q') + f(q')), \upsilon) \mid q, q' \in Q,\, s(q) \neq \_,$ and $(q', \upsilon) \in \delta(q, \sigma)\}$, where $f(q) = 1\, \forall q \in F$, and $f(q) = 0\, \forall q \notin F$, be the function that collects the transitions of $\mathcal{U}$. The transition function δ_∞ is defined as follows: $\delta_\infty(s, \sigma) = \bigwedge_{\upsilon \in \Upsilon}(s_\upsilon, \upsilon)$ with $s_\upsilon(q) = max\{n \in \mathbb{N} \mid ((q, n), \upsilon \in \delta^+_\infty(s, \sigma)\}$ (where $max\{\emptyset\} = \_$).

$\mathcal{D}_c$ is formed by restricting the states of $\mathcal{D}$ to $S_c = Q \to \{\_\} \cup \{0, \ldots, c\}$.

Proof: Let λ be a valid c-bounded annotation of $\mathcal{T} = (T, t_0, \tau, o)$ for $\mathcal{U}$, and let λ_t denote the function with $\lambda_t(q) = \lambda(q, t)$. For two functions $s, s' : Q \to \{\_\} \cup \mathbb{N}$, we write $s \leq s'$ if $s(q) \leq s'(q)$ holds for all $q \in Q$, where $\_$ is the minimal element ($\_ < n$ for all $n \in \mathbb{N}$). We show by induction that $\mathcal{D}_c$ has a run graph $\mathcal{G} = (G, E)$ for $\mathcal{T}$, such that $s \leq \lambda_t$ holds true for all vertices $(s, t) \in G$ of the run graph. For the induction basis, $s_0 \leq \lambda_{t_0}$ holds by definition. For the induction step, let $(s, t) \in G$ be a vertex of $\mathcal{G}$. By induction hypothesis, we have $s \leq \lambda_t$. With the definition of δ^+_∞ and the validity of λ, we can conclude that $((q', n), \upsilon) \in \delta^+_\infty(s, o(t))$ implies $n \leq \lambda_{\tau(t,\upsilon)}(q')$, which immediately implies $s' \leq \lambda_{t'}$ for all successors (s', t') of (s, t) in $\mathcal{G}$.

Let now $\mathcal{G} = (G, E)$ be an accepting run graph of $\mathcal{D}_c$ for $\mathcal{T}$, and let $\lambda(q, t) = max\{s(q) \mid (s, t) \in G\}$. Then λ is obviously a c-bounded annotation. For the validity of λ, $\lambda(q_0, t_0) \in \mathbb{N}$ holds true since $s_0(q_0) \in \mathbb{N}$ is a natural number and $(s_0, t_0) \in G$ is a vertex of $\mathcal{G}$. Also, if a pair (q, t) is annotated with a natural number $\lambda(q, t) = n \neq \_$, then there is a vertex $(s, t) \in G$ with $s(q) = n$. If now $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $((q', n + f(q')), \upsilon) \in \delta^+_\infty(s, o(t))$ holds true, and the υ-successor

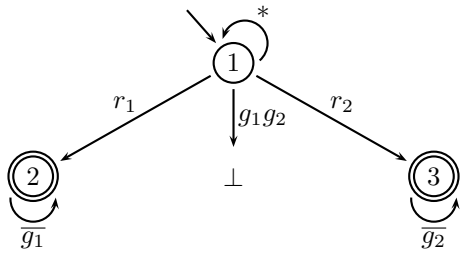


Figure 7.2: Specification of a simple arbiter, represented as a universal Co-Büchi automaton. The states depicted as double circles (2 and 3) are the rejecting states in F.

$(s', \tau(t, \upsilon))$ of (s, t) satisfies $s'(q') \triangleright_{q'} n$. The validity of λ now follows with $\lambda(q', \tau(t, \upsilon) \geq s'(q')$. □

Remark. Since $\mathcal{U}$ may accept transition systems where the number of rejecting states occurring on a path is unbounded, the union of the languages of all $\mathcal{D}_c$ is, in general, a strict subset of the language of $\mathcal{U}$. Every finite-state transition system in the language of $\mathcal{U}$, however, is accepted by almost all $\mathcal{D}_c$.

Example. Consider the specification of a simple arbiter, depicted as a universal Co-Büchi automaton in Figure 7.2. The specification requires that globally

- at most one process has a grant, and
- each request is eventually followed by a grant.

The emptiness game for $\mathcal{D}_1$ intersected with $\mathcal{D}_\mathcal{I}$ is depicted in Figure 7.3.

7.5 Constraint-Based Bounded Synthesis

We now develop an alternative synthesis method for fully informed architectures that uses a SAT solver to determine an input-preserving transition system with a valid annotation. The constraint system defined in this section will provide the foundation for the synthesis method for general distributed architectures in Section 7.6.

We represent the (unknown) transition system and its annotation by uninterpreted functions. The existence of a valid annotation is thus reduced to

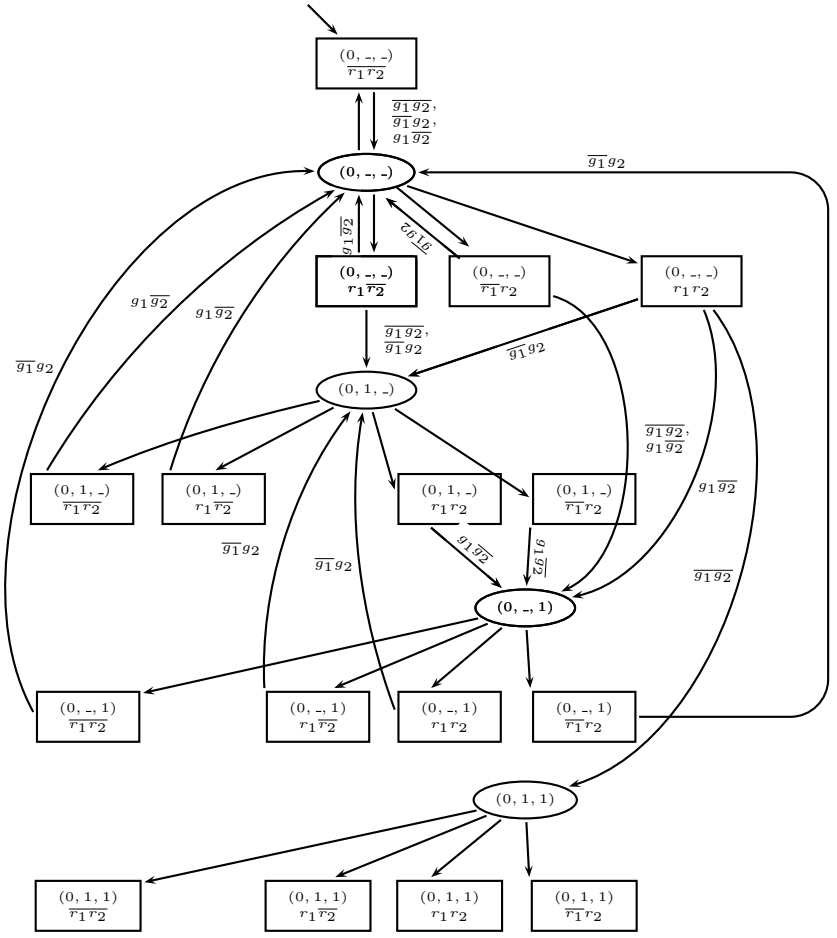


Figure 7.3: Example of a safety game for synthesis in a fully informed architecture. The figure shows the emptiness game for the intersection of $\mathcal{D}_1$ and $\mathcal{D}_{\mathcal{I}}$ in the arbiter example (Figure 7.2). Rectangles denote game positions for player *accept*, ovals denote game positions for player *reject*. Game positions that are not completely expanded (that is, that have more successors if the parameter is increased) are dashed. The starting position specifies $\overline{r_1}\,\overline{r_2}$ as root direction. Player *accept* wins the game by avoiding the move to $(0, 1, 1)$.

the satisfiability of a constraint system in first-order logic modulo finite integer arithmetic. The advantage of this representation is that the size of the constraint system is small (bilinear in the size of $\mathcal{U}$ and the number of directions). Furthermore, the additional constraints needed for distributed synthesis, which will be defined in Section 7.6, have a compact representation as well (logarithmic in the number of directions of the individual processes).

Remark. Integer arithmetic is useful for explaining the algorithm, but we do not need to build on integer arithmetic: The essential property is to guarantee the absence of cycles that contain a rejecting state. But to prove this absence, *any* ordered set (finite or not) suffices for the labels. Finiteness, in turn, is only used to restrict the size of the system. An efficient implementation will therefore build on theories with more efficient algorithms.

The constraint system specifies the existence of a finite input-preserving 2^{Π}-labeled $2^{O_{env}}$-transition system $\mathcal{T} = (T, t_0, \tau, o)$ that is accepted by the universal Co-Büchi automaton $\mathcal{U}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ and has a valid annotation λ.

To encode the transition function τ, we introduce a unary function symbol τ_υ for every output $\upsilon \subseteq O_{env}$ of the environment. Intuitively, τ_υ maps a state t of the transition system $\mathcal{T}$ to its υ-successor $\tau_\upsilon(t) = \tau(t, \upsilon)$.

To encode the labeling function o, we introduce a unary predicate symbol a for every variable $a \in \Pi$. Intuitively, a maps a state t of the transition system $\mathcal{T}$ to *true* if and only if it is part of the label $o(t) \ni a$ of $\mathcal{T}$ in t.

To encode the annotation, we introduce, for each state q of the universal Co-Büchi automaton $\mathcal{U}$, a unary predicate symbol $\lambda_q^{\mathbb{B}}$ and a unary function symbol $\lambda_q^{\#}$. Intuitively, $\lambda_q^{\mathbb{B}}$ maps a state t of the transition system $\mathcal{T}$ to *true* if and only if $\lambda(q,t)$ is a natural number, and $\lambda_q^{\#}$ maps a state t of the transition system $\mathcal{T}$ to $\lambda(q,t)$ if $\lambda(q,t)$ is a natural number, and is unconstrained if $\lambda(q,t) = \_$.

We can now formalize that the annotation of the transition system is valid by the following first order constraints (modulo finite integer arithmetic):

$$\forall t.\, \lambda_q^{\mathbb{B}}(t) \wedge \underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))} \rightarrow \lambda_{q'}^{\mathbb{B}}(\tau_\upsilon(t)) \wedge \lambda_{q'}^{\#}(\tau_\upsilon(t)) \rhd_q \lambda_q^{\#}(t),$$

where $\overrightarrow{a}(t)$ represents the label $o(t)$, $\underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))}$ represents the corresponding propositional formula, and $\rhd_q$ stands for $\rhd_q \equiv >$ if $q \in F$ and $\rhd_q \equiv \geq$ otherwise. Additionally, we require $\lambda_{q_0}^{\mathbb{B}}(0)$, that is, we require the pair of initial states to be labeled by a natural number.

To guarantee that the resulting transition system is input-preserving, we add, for each $a \in O_{env}$ and each $\upsilon \subseteq O_{env}$, a constraint

$$\forall t.\, a(\tau_\upsilon(t)) \text{ if } a \in \upsilon,$$

and a constraint

$$\forall t.\, \neg a(\tau_\upsilon(t)) \text{ if } a \notin \upsilon.$$

Additionally, we require that the initial state is labeled with the root direction.

As an obvious implication of Theorem 7.2, this constraint system is satisfiable if and only if $\mathcal{U}$ accepts a finite input-preserving transition system.

Theorem 7.6 *For fully informed architectures, the constraint system inferred from the specification, represented as the universal Co-Büchi automaton $\mathcal{U}$, is satisfiable modulo finite integer arithmetic if and only if the specification is finite-state realizable.* □

Lemma 7.7 *For a specification represented as a universal Co-Büchi automaton $\mathcal{U} = (2^\Pi, 2^{O_{env}}, Q, q_0, \delta, F)$, the inferred constraint system has size $O(|\delta| \cdot |\Pi| + |O_{env}| \cdot |2^{O_{env}}|)$.* □

The main parameter of the constraint system is the bound b_E on the size of the transition system $\mathcal{T}_E$. If we use b_E to unravel the constraint system completely (that is, if we resolve the universal quantification explicitly), the size of the resulting constraint system is linear in b_E.

Theorem 7.8 *For a specification, represented as a universal Co-Büchi automaton $\mathcal{U} = (2^\Pi, 2^{O_{env}}, Q, q_0, \delta, F)$, and a given bound b_E on the size of the transition system $\mathcal{T}_E$, the unraveled constraint system has size $O(b_E \cdot (|\delta| \cdot |\Pi| + |O_{env}| \cdot |2^{O_{env}}|))$. It is satisfiable if and only if the specification is bounded realizable in the fully informed architecture $(\{env, p\}, \Pi, \{I_p = O_{env}\}, \{O_{env}, O_p = \Pi \setminus O_{env}\})$ with bound b_E.* □

Example. Figure 7.4 shows the constraint system, resulting from the specification of an arbiter by the universal Co-Büchi automaton depicted in Figure 7.2, implemented on the single process architecture of Figure 7.1d (or, likewise, on the distributed but fully informed architecture of Figure 7.1c).

The first constraint represents the requirement that the resulting transition system must be input-preserving, the second requirement represents the initialization (where $\neg r_1(0) \wedge \neg r_2(0)$ represents an arbitrarily chosen root direction), and the requirements 3 through 8 each encode one transition of the universal automaton of Figure 7.2. Following the notation of Figure 7.2, r_1 and r_2 represent the requests and g_1 and g_2 represent the grants.

1. $\forall t.\, r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{r_1 \overline{r}_2}(t))$
   $\wedge \neg r_1(\tau_{\overline{r}_1 r_2}(t)) \wedge r_2(\tau_{\overline{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{\overline{r}_1 \overline{r}_2}(t))$

2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$

3. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 r_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}(t)$

4. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$

5. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$

6. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$

7. $\forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t)$

8. $\forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)$

Figure 7.4: Example of a constraint system for synthesis in a fully informed architecture. The figure shows the constraint system for the arbiter example (Figure 7.2). The arbiter is to be implemented in the fully informed architecture shown in Figure 7.1d.

7.6 Distributed Synthesis

To solve the distributed synthesis problem for a given environment architecture $\mathbf{E} = (B, \Pi, \{I_b\}_{b\in B}, \{O_b\}_{b\in B\cup\{env\}})$, we need to find a family of (finite-state) transition systems $\{\mathcal{T}_b = (T_b, t_0^b, \tau_b, o_b) \mid b \in B\}$ such that their composition to $\mathcal{T}_E$ satisfies the specification. The constraint system developed in the previous section can be adapted to distributed synthesis by explicitly decomposing the global state space of the combined transition system $\mathcal{T}_E$: We introduce a unary function symbol d_b for each process $b \in B$, which, intuitively, maps a state $t \in T_E$ of the product state space to its b-component $t_b \in T_b$.

The value of an output variable $a \in O_b$ may only depend on the state of the process transition system $\mathcal{T}_b$. We therefore replace every occurrence of $a(t)$ in the constraint system of the previous section by $a(d_b(t))$. Additionally, we require that every process b acts consistently on any two histories that it cannot distinguish. The update of the state of $\mathcal{T}_b$ may thus only depend on the state of $\mathcal{T}_b$ and the input visible to b. This is formalized by the following constraints:

1. $\forall t.\, d_b(\tau_\upsilon(t)) = d_b(\tau_{\upsilon'}(t))$ for all decisions $\upsilon, \upsilon' \subseteq O_{env}$ of the environment that are indistinguishable for p (that is, $\upsilon \cap I_b = \upsilon' \cap I_b$).

2. $\forall t, u.\, d_b(t) = d_b(u) \wedge \bigwedge_{a\in I_b \smallsetminus O_{env}} \big(a(d_{b_a}(t)) \leftrightarrow a(d_{b_a}(t))\big) \rightarrow d_b(\tau_\upsilon(t)) = d_b(\tau_\upsilon(u))$ for all decisions $\upsilon \subseteq O_{env} \cap I_b$ (picking one representative for each class of environment decisions that p can distinguish). $b_a \in B$ denotes the process controlling the output variable $a \in O_{b_a}$.

Since the combined transition system $\mathcal{T}_E$ is finite-state, the satisfiability of this constraint system modulo finite integer arithmetic is equivalent to the distributed synthesis problem.

Theorem 7.9 *The constraint system inferred from the specification, represented as the universal Co-Büchi automaton $\mathcal{U}$, and the environment architecture $\mathbf{E}$ is satisfiable modulo finite integer arithmetic if and only if the specification is finite-state realizable in the environment architecture $\mathbf{E}$.* □

Lemma 7.10 *For a specification, represented as a universal Co-Büchi automaton $\mathcal{U} = (2^\Pi, 2^{O_{env}}, Q, q_0, \delta, F)$, and an environment architecture $\mathbf{E}$, the inferred constraint system for distributed synthesis has size $O(|\delta| \cdot |\Pi| + |O_{env}| \cdot |2^{O_{env}}| + \sum_{b\in B} |I_b \smallsetminus O_{env}|)$.* □

4. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_1(t)) \vee \neg g_2(d_2(t))$

7. $\forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_1(t)) \rightarrow \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_2^{\#}(t)$
$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t)$
$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_2^{\#}(t)$
$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t)$

8. $\forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_2(t)) \rightarrow \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_3^{\#}(t)$
$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t)$
$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_3^{\#}(t)$
$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)$

9. $\forall t.\, d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1\overline{r}_2}(t)) \wedge d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1\overline{r}_2}(t))$
$\wedge\ d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{\overline{r}_1 r_2}(t)) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t))$

10. $\forall t, u.\, d_1(t) = d_1(u) \wedge \big(g_2(d_2(t)) \leftrightarrow g_2(d_2(u))\big)$
$\rightarrow d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1 r_2}(u)) \wedge d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1 r_2}(u))$

11. $\forall t, u.\, d_2(t) = d_2(u) \wedge \big(g_1(d_1(t)) \leftrightarrow g_1(d_1(u))\big)$
$\rightarrow d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{r_1 r_2}(u)) \wedge d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{r_1\overline{r}_2}(u))$

Figure 7.5: Example of a constraint system for distributed synthesis. The figure shows modifications and extensions to the constraint system from Figure 7.4 for the arbiter example (Figure 7.2) in order to implement the arbiter in the distributed architecture shown in Figure 7.1b.

The main parameters of the constraint system for distributed synthesis are the bound b_E on the size of the transition system $\mathcal{T}_E$ and the family $\{b_b \mid b \in B\}$ of bounds on the process transition systems $\{\mathcal{T}_b \mid b \in B\}$. If we use these parameters to unravel the constraint system completely (that is, if we resolve the universal quantification explicitly), the resulting transition system is linear in b_E, and quadratic in b_b.

Theorem 7.11 *For a given specification, represented as a universal Co-Büchi automaton* $\mathcal{U} = (2^{\Pi}, 2^{O_{env}}, Q, q_0, \delta, F)$*, an environment architecture* $\mathbf{E} = (B, \Pi, \{I_b\}_{b\in B}, \{O_b\}_{b\in B\cup\{env\}})$*, a bound* b_E *on the size of the input-preserving transition system* $\mathcal{T}_E$*, and a family* $\{b_b \mid b \in B\}$ *of bounds on the process transition systems* $\{\mathcal{T}_b \mid b \in B\}$*, the unraveled constraint system has size* $O(b_E \cdot (|\delta| \cdot |\Pi| + |O_{env}| \cdot |2^{O_{env}}|) + \sum_{b\in B} {b_b}^2 |I_b \smallsetminus O_{env}|))$*. It is satisfiable if*

bound	4	5	6	7	8	9
result	unsatisfiable	unsatisfiable	unsatisfiable	unsatisfiable	satisfiable	satisfiable
# decisions	3957	13329	23881	68628	72655	72655
# conflicts	209	724	1998	15859	4478	4478
# Boolean variables	1011	2486	4169	9904	5214	5214
memory (MB)	16.9102	18.1133	20.168	27.4141	26.4375	26.4414
time (seconds)	0.05	0.28	1.53	35.99	7.53	7.31

Table 7.1: Experimental results from the synthesis of a single-process arbiter using the specification from Figure 7.2 and the architecture from Figure 7.1a. The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Figure 7.4, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

and only if the specification is bounded realizable in A for the bounds b_E and $\{b_b \mid b \in B\}$. □

Example. As an example for the reduction of the distributed synthesis problem to SAT, we consider the problem of finding a distributed implementation to the arbiter specified by the universal automaton of Figure 7.2 in the architecture of Figure 6.3b. The functions d_1 and d_2 are the mappings to the processes p_1 and p_2, which receive requests r_1 and r_2 and provide grants g_1 and g_2, respectively. Figure 7.5 shows the resulting constraint system. Constraints 1–3, 5, and 6 are the same as in the fully informed case (Figure 7.4). The consistency constraints 9–11 guarantee that processes p_1 and p_2 show the same behavior on all input histories they cannot distinguish.

7.7 Experimental Results

Using the reduction described in the previous sections, we considered five benchmarks; we synthesized implementations for simple arbiter specification from Figure 7.2 and the two architectures from Figure 7.1, and for a full arbiter specification and the two architectures from Figure 7.1, and we synthesized a strategy for dining philosophers to satisfy the specification from Figure 7.7. The arbiter examples are parametrized in the size of the transition system(s), the dining philosophers benchmark is additionally parametrized in the number of philosopher. As the SMT solver, we used Yices version 1.0.9 on a 2.6 Ghz Opteron system.

In all benchmarks, Yices is unable to directly determine the satisfiability of the quantified formulas. (For example the formulas from Figure 7.4 and Fig-

bound	4	5	6	7
result	unsatisfiable	unsatisfiable	unsatisfiable	unsatisfiable
# decisions	6041	15008	35977	89766
# conflicts	236	929	2954	30454
# Boolean variables	1269	2944	5793	9194
memory (MB)	17.0469	18.4766	22.1992	33.1211
time (seconds)	0.06	0.35	3.3	120.56

bound	8	9	8 (1)	8 (2)
result	satisfiable	satisfiable	unsatisfiable	satisfiable
# decisions	197150	154315	178350	71074
# conflicts	33496	24607	96961	18263
# Boolean variables	7766	8533	12403	6382
memory (MB)	37.4297	36.2734	39.4922	29.1992
time (seconds)	70.97	58.43	200.07	36.38

Table 7.2: Experimental results from the synthesis of a two-process arbiter using the specification from Figure 7.2 and the architecture from Figure 7.1b. The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Figure 7.5, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in parentheses).

bound	4	5	6	7	8
result	unsatisfiable	satisfiable	satisfiable	satisfiable	satisfiable
# decisions	17566	30011	52140	123932	161570
# conflicts	458	800	1375	2614	3987
# Boolean variables	1850	2854	3734	5406	6319
memory (MB)	18.3008	20.0586	22.5781	27.5000	35.7148
time (seconds)	0.21	0.63	1.72	5.15	12.38

Table 7.3: Experimental results from the synthesis of a single-process arbiter using the specification from Figure 7.6 and the architecture from Figure 7.1a. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

ure 7.5, respectively, for the monolithic and distributed synthesis in the simple arbiter example.) However, after replacing the universal quantifiers with explicit conjunctions (for a given upper bound on the number of states in the implementation), Yices solved all satisfiability problems quickly.

A single-process implementation of the arbiter needs 8 states. Table 7.1 shows the time and memory consumption of Yices when solving the SMT problem from Figure 7.4 with the quantifiers unraveled for different upper bounds on the number of states. The correct implementation with 8 states is found in 8 seconds.

7.7.1 Arbiter

Table 7.2 shows the time and memory consumption for the distributed synthesis problem. The quantifiers in the formula from Figure 7.5 were unraveled for different bounds on the size of the global transition system and for different bounds (shown in parentheses) on the size of the processes. A correct solution with 8 global states is found by Yices in 71 seconds if the number of process states is left unconstrained. Restricting the process states explicitly to 2 leads to an acceleration by a factor of two (36 seconds).

Table 7.3 and Table 7.4 show the time and memory consumption of Yices when solving the SMT problem resulting from the arbiter specification of Figure 7.6. The correct monolithic implementation with 5 states is found in less than one second, and Yices needs only half a minute to construct a correct distribute implementation. The table also shows that borderline cases like the

bound	4	5	6	7	8	9
result	unsat	unsat	unsat	unsat	sat	sat
# decisions	16725	47600	91480	216129	204062	344244
# conflicts	326	1422	8310	61010	11478	16347
# Boolean variables	1890	7788	5793	13028	8330	10665
memory (MB)	18.0273	22.2109	28.5312	43.8594	42.2344	61.9727
time (seconds)	0.16	1.72	14.84	208.78	32.47	72.97

bound	8 (1)	8 (2)	8 (3)	8 (4)
result	unsat	unsat	sat	sat
# decisions	309700	1122755	167397	208255
# conflicts	92712	775573	13086	13153
# Boolean variables	15395	25340	8240	7806
memory (MB)	54.1641	120.0160	42.1484	42.7188
time (seconds)	263.44	5537.68	31.12	30.36

Table 7.4: Experimental results from the synthesis of a two-process arbiter using the specification from Figure 7.6 and the architecture from Figure 7.1b. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in parentheses).

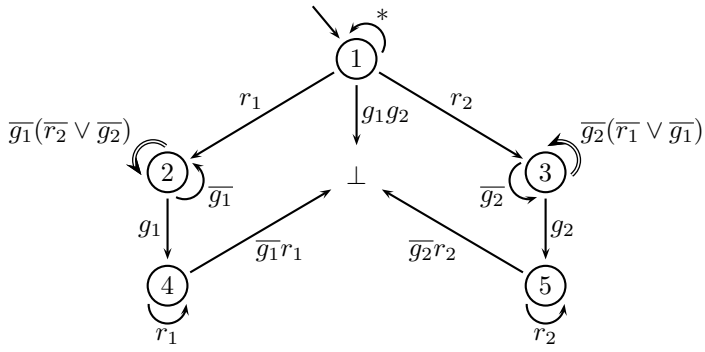


Figure 7.6: Extended specification of an arbiter, represented as a universal Co-Büchi automaton with edge-based acceptance. (Since our acceptance mechanism is edge based, it is more efficient to use an edge-based acceptance condition for the automaton. Moving from state-based to edge-based acceptance can lead to a reduction in the statespace up to 50%.) The edges depicted as double-line arrows are the rejecting edges in F.

fruitless search for an implementation with 8 states, but only 2 local states, can become very expensive; in the example, Yices needed more than 1.5 hours to determine unsatisfiability. Compromising on optimality, by slightly increasing the bounds, greatly improves the performance. Searching for an implementation with 8 states and 3 or 4 local states takes approximately 30 seconds.

7.7.2 Dining Philosophers

Table 7.5 shows the time and memory consumption for synthesizing a strategy for the dining philosophers to satisfy the specification shown in Figure 7.7. In the dining philosophers benchmark, the size of the specification grows linearly with the number of philosophers; for 10.000 philosophers this results in systems of hundreds of thousands constraints. In spite of the large size of the resulting constraint system, the synthesis problem remains tractable; Yices solves all resulting constraint systems within a few hours, and within a minutes for small constraint systems with up to 1000 philosophers.

	3 states			4 states			6 states		
# phil.	time (s)	memory (MB)	result	time (s)	memory (MB)	result	time (s)	memory (MB)	result
125	1.52	23.2695	unsat	23.84	36.2305	unsat	236.5	87.7852	sat
250	5.41	29.2695	unsat	130.07	52.0859	sat	141.36	91.1328	sat
375	22.81	38.9727	unsat	128.83	58.1992	unsat	890.58	154.355	sat
500	17.98	39.9297	unsat	15.84	52.9336	sat	237.04	119.309	sat
625	35.57	49.5586	unsat	417.05	94.7188	unsat	486.5	130.977	sat
750	22.25	52.3359	unsat	20.85	69.1562	sat	82.63	99.707	sat
875	51.98	56.0859	unsat	628.84	119.363	unsat	2546.88	255.965	sat
1000	168.17	70.3906	unsat	734.74	117.703	sat	46.18	124.691	sat
1125	67.14	70.1133	unsat	1555.18	165.922	unsat	1854.77	246.848	sat
1250	165.59	76.2227	unsat	122.8	107.645	sat	596.8	203.012	sat
1375	104.27	75.4531	unsat	3518.85	191.113	unsat	8486.18	490.566	sat
1500	187.25	82.8867	unsat	85.52	129.215	sat	232.81	214.68	sat
1625	85.83	88.8047	unsat	2651.82	246.734	unsat	1437.45	281.203	sat
1750	169.93	97.543	unsat	107.14	126.477	sat	257.77	185.887	sat
1875	174.03	105.25	unsat	3629.18	234.527	unsat	4641.03	405.781	sat
2000	25.86	102.125	unsat	242.55	157.734	sat	811.78	269.375	sat
2125	163.39	113.27	unsat	5932.24	315.711	unsat	6465.75	424.121	sat
2250	412.37	115.438	unsat	523.87	162.391	sat	5034.83	456.316	sat
2375	201.95	120.047	unsat	7311.03	313.168	unsat	4887.76	451.332	sat
2500	375.29	135.535	unsat	235.17	202.59	sat	319.78	253.781	sat
2625	544.03	135.379	unsat	6560.53	312.355	unsat	23990.5	808.633	sat
2750	559.35	139.137	unsat	817.41	226.082	sat	632.28	349.992	sat
2875	308.36	151.727	unsat	7273.89	299.016	unsat	8638.96	551.5	sat
3000	666.18	155.57	unsat	533.23	228.961	sat	3158.26	493.617	sat
3125	235.52	141.93	unsat	12596.6	377.328	unsat	10819.7	693.133	sat
3250	869.53	153.633	unsat	2089.72	308.719	sat	21298.8	889.285	sat
3375	260.88	145.918	unsat	11581.7	379.949	unsat	21560	741.09	sat
3500	308.23	169.348	unsat	897.6	270.676	sat	829.52	398.008	sat
5000	982.68	240.273	unsat	3603.7	421.832	sat	1357.48	582.457	sat
7000	2351.87	313.277	unsat	7069.55	535.98	sat	6438.73	1081.68	sat
10000	4338.83	448.648	unsat	4224.28	761.008	sat	10504.6	1121.58	sat

Table 7.5: Experimental results from the synthesis of a strategy for the dining philosophers using the specification from Figure 7.7. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

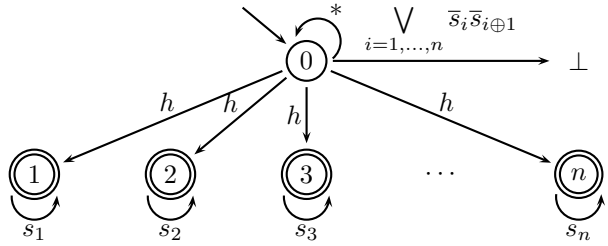


Figure 7.7: Specification of a dining philosopher problem with n philosophers. The environment can cause the philosophers to become hungry (by setting h to true). The states depicted as double circles (1 through n) are the rejecting states in F; state i refers to the situation where philosopher i is hungry and starving (s_i). A fail state is reached when two adjacent philosophers try to reach for their common chopstick; the fail state refers to the resulting eternal philosophical quarrel that keeps the affected philosophers from eating.

7.8 Extension to General Architectures and ATμC Specification

The restrictions to linear-time logic and environment architectures is only chosen for two reasons: First and foremost, they simplify the presentation, but it seems also doubtful that more general specification languages can be treated comparably efficiently.

To extend the methods proposed in this chapter to general architectures and ATμC specifications, we first observe that the synthesis technique proposed in Chapter 6 produces an input enabled transition system, namely the finite state Moore machine from Theorem 6.11. Instead of constructing it using the purely automata based approach from Chapter 6, we can guess this transition system, and instead of using the quotient construction based on finding a coarsest bisimulation relation for the extraction of the single implementations, we can use the constraints from Section 7.6 to guarantee that the processes only use information available to them.

Technically, these extensions are simple:

- We start with constructing an alternating automaton $\mathcal{B}$ that accepts a relaxed implementation $\langle (2^\Pi)^*, l \times \bigoplus_{a \in A} p'_a \rangle$ (extended by atomic propositions; $l : (2^\Pi)^* \to 2^\Pi$) with input Π if and only if $\langle \|p'_A\|, l \rangle$ is a model of φ (Theorem 4.1, and Lemmata 4.2 and 6.4, cf. Subsection 6.5.1).

- If $\mathcal{B} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ accepts an input enabled transition system $\mathcal{T} = (T, t_0, \tau, o)$, than it has a memoryless winning strategy in the acceptance game. This memoryless winning strategy can be viewed as a family of mappings $\{s_t : Q \to 2^{Q \times \Pi} \mid t \in T\}$ from states of the automaton to sets of successor states and directions, one for each state. Since such a mapping exists, we can require that s_t is explicitly represented in the label $o(t)$.

  Once the strategy is fixed, $\mathcal{B}$ behaves like a universal automaton. Thus we obtain a universal automaton $\mathcal{C} = (\Sigma \times (Q \to 2^{Q \times \Pi}), \Upsilon, Q, q_0, \delta, \alpha)$ that accepts, for every bounded transition system $\mathcal{T}$ accepted by $\mathcal{B}$, a bounded transition system $\mathcal{T}'$ with the same family of bounds, such that $\mathcal{T}$ is the projection of $\mathcal{T}'$ to Σ.

- A universal parity automaton $\mathcal{C} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ with odd colors $C_o = \alpha(Q) \setminus 2\mathbb{N}$ can be translated into an equivalent universal Co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q \cup Q \times C_0, q_0, \delta', F)$ with $F = \{(q, c) \mid \alpha(q) = c$ by choosing

  - $\delta'(q, \sigma) = b_q^\sigma\big(\{(q_i, \upsilon_i) \wedge \bigwedge_{c \in C_o} ((q_i, c), \upsilon_i) \mid i \in I_q^\sigma\}\big)$, and
  - $\delta'\big((q, c), \sigma\big) = b_q^\sigma\big(\{\bigwedge_{c' \in C_o, c' \leq c} ((q_i, c'), \upsilon_i) \mid i \in I_q^\sigma\}\big)$,

  where $b_q^\sigma\big(\{(q_i, \upsilon_i) \mid i \in I\}\big) = \delta(q, \sigma)$ is the positive Boolean function defined by $\delta(q, \sigma)$. (Note that the empty conjunction is *true*.) Intuitively, $\mathcal{D}$ can guess any point in the run, where the highest color occurring in the future *and* infinitely often is some odd color $c \in C_o$, and move to the c copy at that point in the run.

$\mathcal{U}$ can, instead of the automaton constructed by Theorem 7.1, be used as the starting point for the algorithm discussed in this chapter, and white-box implementations can be explicitly represented.

7.9 Conclusions

Despite its obvious advantages, synthesis has been less popular than verification. While the complexity of verification is determined by the size of the implementation under analysis, standard synthesis algorithms [PR90, KV01, MT01, WM03] for distributed systems – including the synthesis algorithm introduced in Chapter 6 – suffer from the daunting complexity determined by the theoretical upper

bound on the smallest implementation, which, as shown by Rosner [Ros92], increases by an extra exponent with each additional process in the architecture.

By introducing a bound on the size of the implementation, we have leveled the playing field for synthesis and verification. We have shown that the bounded synthesis problem can be solved effectively with a reduction to SAT.

Our solution for the bounded synthesis problem can be extended to the standard (unbounded) synthesis problem by iteratively increasing the bound. The advantage of this approach is that the complexity is determined by the size of the smallest implementation. Typically, this implementation is far smaller than the exploding upper bound.

Chapter 8

Excursion: Probabilistic Environments

Abstract

In synthesis, we construct finite state systems from their temporal specifications. While this problem is well understood in the classical setting of non-probabilistic synthesis, this chapter suggests an extension of open synthesis techniques to a setting, where the environment chooses its actions randomized rather than nondeterministically. Assuming a randomized environment inspires alternative semantics both for linear-time and branching-time logics. For linear-time, natural acceptance criteria are *almost-sure* and *observable* acceptance, where it suffices if the probability measure of accepting paths is 1 and greater than 0, respectively. Almost-sure and observable semantics for linear-time logic also suggest an alternative interpretation of branching-time specifications, where existential quantification is identified with observable acceptance and universal quantification with almost-sure acceptance.

We distinguish 0-environments, which can freely assign probabilities to each environment action, from ε-environments, where the probabilities assigned by the environment are bounded from below by some $\varepsilon > 0$. While the results in case of 0-environments are essentially the same as for nondeterministic environments, the languages occurring in case of ε-environments are topologically different from the results for nondeterministic and 0-environments (in case of LTL, recognizable by weak alternating automata vs. recognizable by determin-

istic automata). The complexity of open synthesis is, in both cases, EXPTIME-complete and 2EXPTIME-complete for CTL and LTL specifications, respectively.

8.1 Introduction

In the view of the attractiveness of synthesis, it is alluring to extend its applicability as far as possible. A particular interesting extension is the treatment of probabilistic (or: randomized) systems. Randomization has, for example, successfully been introduced into protocols (cf. [LR81]). In synthesis, we want to construct systems which, under reasonable assumptions about the probabilistic behavior of the environment, satisfy a linear-time specification with probability 1 (*almost-surely*) or with probability greater than 0 (*observably*).

System synthesis is harder than model-checking probabilistic systems (Markov decision processes). There, a probabilistic measure is defined a priori on the set of computations, usually by assigning fixed probabilities to the single transitions. In synthesis, on the other hand, we do not have a transition-system to start with (this situation is comparable with the problem occurring in the treatment of transition fairness in system synthesis, cf. [AM94]).

When restricting the scope to almost-sure and observable satisfaction of linear-time properties, the concrete probabilities of single transitions play a minor role; in finite systems it is only of interest whether or not a probability is 0 or 1. It turns out that these properties are preserved when the probabilities of the single transitions are uncertain, as long as an (arbitrary) lower bound $\varepsilon > 0$ on their probability is guaranteed. This allows for considering synthesis for environments, which only guarantee the *existence* of some lower bound on the probability of each single action. We call such environments *ε-environments*. They are closely related to probabilistic fair systems [dA99] (with the distinction that systems discussed in this chapter necessarily have a predefined *constant* set of environment actions) and inherit their semantic benefits: They provide a simple way of representing probabilistic choices while abstracting from the numerical value of probability. The LTL synthesis problem remains 2EXPTIME-complete in almost-sure and observable semantics for ε-environments.

The decidability of almost-sure and observable acceptance gives rise to a re-definition of the semantics for the branching-time logic CTL*. CTL* allows for universal ($A\pi$) and existential ($E\pi$) path quantification. A natural analogy is to interpret universal path quantification as the property that the probability measure of the paths satisfying π is 1 (that is, that a path almost-surely satisfies

π), and existential path quantification as the property that the probability measure of the paths satisfying π is greater than 0 [HJ94]. This chapter provides a constructive method to solve the synthesis problem for CTL* in 3EXPTIME in the length of the specification, whereas a 2EXPTIME lower bound is inherited from the LTL synthesis problem. While the exact complexity remains open for CTL*, the synthesis problem is EXPTIME-complete for CTL.

Under the assumption of stronger environments, which can reduce the probability of each single event arbitrarily, synthesis for almost-sure/observable semantics is essentially equivalent to synthesis for classical semantics.

8.2 Preliminaries

Synthesis algorithms automatically construct, for a given class of environments, systems that are correct by construction from a given specification. The environment is an external part of the system, which is not under the control of the synthesis algorithm. Intuitively, the environment provides the system with inputs from a finite input-alphabet Υ. The system reacts on each input by emitting an output symbol from a finite output-alphabet Σ. When the specifications are provided as temporal logics, the input- and output alphabet consist of the possible valuations of Boolean input- and output-variables, respectively [PR89a, KV97b, KV99], which also serve as atomic propositions in the specification. A system is modeled as a finite transition-system, which defines a mapping $s : \Upsilon^* \to \Sigma$ from histories of input-signals to output-signals. This chapter addresses synthesis for linear- and branching-time specifications for environments with an uncertain probabilistic behavior.

8.2.1 Probabilistic Environments

In general, the concrete behavior of the environment is unknown or too complex to represent. The uncertainty with respect to the concrete behavior of the environment is expressed by the power of the environment to choose, in every step, a probability distribution of its single input letters.

An environment is called an *ε-environment* if, in each step, the probability $p(\upsilon) \in [\varepsilon, 1]$ that the environment chooses a particular input letter $\upsilon \in \Upsilon$ is bound from below by some $\varepsilon > 0$. It is called a *0-environment*, if the probability that the environment chooses a particular input letter $\upsilon \in \Upsilon$ is not bound from below ($p(\upsilon) \in]0, 1]$ or $p(\upsilon) \in [0, 1]$).

8.2.2 The Synthesis Problem

For trace languages, we distinguish *almost-sure* and *observable* acceptance of transition-systems. A transition-system $\mathcal{T}$ satisfies a specification

- *almost-surely* if and only if the probability measure of the set of infinite paths defined by $\mathcal{T}$ that satisfy the specification is 1, and
- *observably* if and only if the probability measure of the set of infinite paths defined by $\mathcal{T}$ that satisfy the specification is greater than 0.

In case of temporal logics, the input-alphabet 2^I and output-alphabet 2^O represent the possible assignments to Boolean input and output variables, which also serve as atomic propositions in the specification.

For CTL* specifications, all subformulas of the form $A\pi$ and $E\pi$ are interpreted as state formulas with the semantics that the path formula π is satisfied almost-surely and observably, respectively. The *synthesis problem* is to either construct, for a given input-alphabet Υ, a given output-alphabet Σ and a specification φ, an input-preserving $\Upsilon \times \Sigma$-labeled Υ-transition-system which satisfies the specification, or to prove that no such transition-system exists.

8.3 Synthesis for Trace Languages

Following an automata-theoretic approach to open synthesis, the synthesis problem is decomposed into two parts: Finding an automaton, which accepts a transition-system if and only if it is input-preserving and satisfies the specification, and constructing a transition-system accepted by this automaton (or demonstrating its emptiness). In this section, we consider synthesis for specifications provided as deterministic word automata under the assumption of ε-environments.

8.3.1 Structural Acceptance Criteria

Testing whether a transition-system $\mathcal{T}$ almost-surely (observably) satisfies a deterministic word automaton $\mathcal{D}$ can be reduced to a simple structural argument over the composition of $\mathcal{T}$ and $\mathcal{D}$. The result of their composition is a colored graph, and it suffices to check if the highest color in all (some) reachable strongly connected components of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ that are leaves in the SCC-graph of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is even.

The composition $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}} = \mathcal{T} \| \mathcal{D}$ of a transition-system $\mathcal{T} = (S, s_0, \tau, l)$ and a deterministic word automaton $\mathcal{D} = (\Sigma, Q, q_0, \delta, \alpha)$ is a colored graph

$\mathcal{G}_{\mathcal{D}}^{\mathcal{T}} = (S \times Q, (s_0, q_0), \tau', \alpha')$ with transition function $\tau' : ((s,q),\upsilon) \mapsto (\tau(s,\upsilon), \delta(q, l(s)))$ and coloring function $\alpha' : (s,q) \mapsto \alpha(q)$.

Lemma 8.1 *An Υ-transition-system $\mathcal{T}$ almost-surely (observably) satisfies a specification provided as a deterministic word automaton $\mathcal{D}$ if and only if the highest color in all (some) reachable leaf-SCCs of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}} = \mathcal{T} \| \mathcal{D}$ is even.*

Proof: For all ε-environments, the probability of every single transition is bounded from below by some $\varepsilon \in]0,1]$. This implies the following attributes of the computations:

- Almost-surely almost all states of a computation are in a single leaf of the SCC-tree of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$, which is reachable from the initial state of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$:
  If $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ has n states, then, from every state of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$, the probability *not* to reach some leaf-SCC within the next n steps is bounded from above by $\varepsilon' = 1 - \varepsilon^n < 1$, which implies a probability of 0 to stay forever out of reachable leaf-SCCs.

- Every reachable leaf-SCC of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is reached with some positive probability (which is bounded from below by ε^n).

- For traces that eventually reach a leaf-SCC L, the highest color occurring infinitely often is almost-surely the highest color of the states of L:
  The probability *not* to reach some state s in L within the next n steps is again bounded from above $\varepsilon' = 1 - \varepsilon^n < 1$. This implies, for every position in the trace, a probability of 0 that s occurs never again; this holds in particular for a state s whose color is maximal in L.

Thus, the probability measure of the paths that satisfy the the specification is 1 if and only if the highest color in all reachable leaf SCCs is even, and > 0 if and only if the highest color in at least one reachable leaf SCC is even. □

8.3.2 Game Construction

These structural criteria can be transformed into weak *acceptance games* deciding almost-sure and observable acceptance, respectively. These games are played on $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$, starting in (s_0, q_0), and consist of three phases. For almost-sure (observable) acceptance, the game is played according to the following rules:

- In the first phase, player *reject* (*accept*) either chooses to proceed to the second phase or picks a transition in $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$. Picking a transition means that,

in a state (s, q), she chooses a direction v and the game proceeds in $\tau'((s, q), v)$.
Intuitively, she can use this phase to move to a leaf-SCC of her choice.

- In the second phase, player *accept* (*reject*) either picks a transition in $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ or chooses to proceed to the third phase, but with the restriction that he can only move to the third phase if the color of the current node is even (odd). In case he moves to the third phase, the color c of the current node is stored.
  This phase is to prevent player *reject* (*accept*) from "cheating" by terminating the first phase in a state of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$, which is not an element of any leaf-SCC. Player *accept* could, in such a case, move on to a vertex with highest color in a leaf-SCC of his choice (reachable from v), or even pick any arbitrary state reachable from v.

- In the last phase, player *reject* (*accept*) again chooses the transitions. She wins immediately upon reaching a state with an odd (even) color greater than c.

Infinite plays of the game are won by player *accept* (*reject*) if the game always stays in the first phase and if the game eventually stays forever in the third phase, while player *reject* (*accept*) wins if the game eventually stays forever in the second phase.

Lemma 8.2 *The acceptance game on $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is won by player* accept *if and only if $\mathcal{T}$ satisfies $\mathcal{D}$ almost-surely (observably).*

Proof: To prove the claim for almost-sure acceptance, first assume that $\mathcal{T}$ does not satisfies $\mathcal{D}$ almost-surely. In this case, the highest color in some reachable leaf-SCC L of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is odd by Lemma 8.1. Player *reject* can direct the game towards such a leaf-SCC L and then let the game proceed to the second phase.

If player *accept* ever moves on to the third phase, he must do so from a state in L. Since L is strongly connected, player *reject* can then move to a state with maximal (odd) color and wins directly. If, on the other hand, player *accept* never moves to the third phase, player reject wins since the third phase is never reached.

To prove the "if" direction, recall that almost-sure satisfaction of $\mathcal{D}$ by $\mathcal{T}$ entails that the highest color in all reachable leaf-SCCs of $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is even. If player *reject* never leaves the first phase, player *accept* wins due to the winning condition for infinite plays. If player *reject* eventually changes in some state v to

the second phase, then player *accept* can move to *some* leaf-SCC L. Since L is strongly connected by definition, he can reach a state v' in L, whose (even) color is maximal in L. After having moved on to v', player *accept* changes to the third phase (storing the color of v'). Since the color of v' is maximal in L, player *reject* cannot win directly in the third phase, and consequently loses by the winning condition for infinite plays.

The proof for observable acceptance runs accordingly. □

From Acceptance Games to Automata. It is only a small step from the acceptance games of the previous paragraph to weak alternating automata over transition-systems. A given deterministic word automaton $\mathcal{D}$ can be turned into weak alternating automata, which accept a transition-system if and only if it satisfies $\mathcal{D}$ almost-surely or observably, respectively. The states of these automata are constructed from the states and colors of $\mathcal{D}$, and the transition function reflects the transitions of the game introduced in the previous paragraph.

Theorem 8.3 *Given a deterministic word automaton $\mathcal{D} = (\Sigma, Q, q_0, \delta, \alpha)$, we can construct weak alternating tree automata $\mathcal{A}_\mathcal{D}$ and $\mathcal{O}_\mathcal{D}$ which accept a Σ-labeled Υ-transition-system if it almost-surely and observably satisfies $\mathcal{D}$, respectively. If $\mathcal{D}$ has n states and c colors, $\mathcal{A}_\mathcal{D}$ and $\mathcal{O}_\mathcal{D}$ have at most $n \cdot \lceil 2 + \frac{c}{2} \rceil$ states.*

Proof: $\mathcal{A}_\mathcal{D} = (\Sigma, \Upsilon, Q', q_0', \delta', \alpha')$ is defined as follows:

- The set of states is set to $Q' = Q \times (\{f, s\} \cup C_e)$ and initial state $q_0' = (q_0, f)$, where C_e denotes the set of even colors of $\mathcal{D}$.
- The transition function is defined by:
  - $\delta' : ((q, f), \sigma) \mapsto \delta'((q, s), \sigma) \wedge \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), f), \upsilon)$,
  - $\delta' : ((q, s), \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon}((\delta(q, \sigma), s), \upsilon)$ if $\alpha(q)$ is odd and
  - $\delta' : ((q, s), \sigma) \mapsto \delta'((q, \alpha(q)), \sigma) \vee \bigvee_{\upsilon \in \Upsilon}((\delta(q, \sigma), s), \upsilon)$ if $\alpha(q)$ is even,
  - $\delta' : ((q, c), \sigma) \mapsto \mathit{false}$ if $\alpha(q)$ is an odd number greater then c, and
  - $\delta' : ((q, c), \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), c), \upsilon)$ otherwise.
- The coloring function α' maps $Q \times \{f\}$ to 0, $Q \times \{s\}$ to 1, and $Q \times C_e$ to 2.

Likewise, $\mathcal{O}_\mathcal{D} = (\Sigma, \Upsilon, Q'', q_0'', \delta'', \alpha'')$ is defined as follows:

- The set of states is set to $Q'' = Q \times (\{f, s\} \cup C_o)$ and initial state $q_0'' = (q_0, f)$, where C_o denotes the set of odd colors of $\mathcal{D}$.
- The transition function is defined by:
  - $\delta'' : ((q, f), \sigma) \mapsto \delta''((q, s), \sigma) \vee \bigvee_{\upsilon \in \Upsilon}((\delta(q, \sigma), f), \upsilon)$,
  - $\delta'' : ((q, s), \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), s), \upsilon)$ if $\alpha(q)$ is even and
  - $\delta'' : ((q, s), \sigma) \mapsto \delta''((q, \alpha(q)), \sigma) \wedge \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), s), \upsilon)$ if $\alpha(q)$ is odd,
  - $\delta'' : ((q, c), \sigma) \mapsto \mathit{true}$ if $\alpha(q)$ is an even number greater then c, and
  - $\delta'' : ((q, c), \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), c), \upsilon)$ otherwise.
- The coloring function α'' maps $Q \times \{f\}$ to 1, $Q \times \{s\}$ to 2, and $Q \times C_o$ to 3.

The states $Q \times \{f\}$ refer to the first phase of the acceptance game, the states $Q \times \{s\}$ to the second and the remaining states $Q \times C_e$ and $Q \times C_o$, respectively, refer to the third phase of the acceptance game. A winning strategy for either player in the acceptance game on $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ can easily be transformed into a winning strategy in the acceptance game of the respective alternating automaton. □

Efficient Nondeterminization. Weak alternating automata are well suited for model-checking, but synthesis (or its non-constructive equivalent, checking non-emptiness) usually contains an exponential blow-up due to a nondeterminization step. A closer look on the special weak alternating automata of Theorem 8.3 reveals that this is not the case here: Most decisions can be guessed by a nondeterministic automaton. The crucial point in the nondeterminization is the single decision of player *reject* when to proceed from the first to the second phase (in case of almost-sure acceptance) and from the second to the third phase (in case of observable acceptance), respectively. It turns out that this single decision can be left uncertain in the construction of a nondeterministic automaton, avoiding the blow-up.

Theorem 8.4 *For a given deterministic word automaton $\mathcal{D} = (\Sigma, Q, q_0, \delta, \alpha)$ we can construct nondeterministic Büchi tree automata $\mathcal{A}_{\mathcal{D}}{}'$ and $\mathcal{O}_{\mathcal{D}}{}'$ which accept a Σ-labeled Υ-transition-system if it almost-surely and observably satisfies $\mathcal{D}$, respectively. If $\mathcal{D}$ has n states and c colors, $\mathcal{A}_{\mathcal{D}}{}'$ and $\mathcal{O}_{\mathcal{D}}{}'$ have at most $2n \cdot \lfloor 1 + \frac{c}{2} \rfloor + 1$ and $n \cdot \lfloor 2 + \frac{c}{2} \rfloor$ states, respectively.*

Proof: The starting points for our construction are the weak alternating automata $\mathcal{A}_\mathcal{D}$ and $\mathcal{O}_\mathcal{D}$ from the proof of Theorem 8.3. Recall that the states of these automata consist of two elements, a state from $\mathcal{D}$ and an indicator for the first, second or third phase (including a stored color for in case of the third phase).

The nondeterministic Büchi tree automaton $\mathcal{O}_\mathcal{D}{}' = (\Sigma, \Upsilon, Q'', q_0'', \delta'', \alpha'')$ for testing observable acceptance is defined as follows:

- The set of states is set to $Q'' = Q \cup Q \times C_o^-$ and the initial state $q_0'' = q_0$ is the initial state from $\mathcal{D}$. C_o^- denotes the set of odd colors of $\mathcal{D}$, plus an additional color $e_{min} = o_{min} - 1$, where o_{min} denotes the smallest odd color of $\mathcal{D}$.
- The transition function is defined by:
  - $\delta'' : (q, \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon}(\delta(q, \sigma), \upsilon) \vee \delta''(q, e_{min}), \sigma)$,
  - $\delta'' : ((q, c), \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon}\Big(((\delta(q, \sigma), \max\{c, \alpha(q)\}), \upsilon)$
    $\wedge \bigwedge_{\upsilon \neq \upsilon' \in \Upsilon}((\delta(q, \sigma), e_{min}), \upsilon)\Big)$ if $\alpha(q)$ is odd,
  - $\delta'' : ((q, c), \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon}((\delta(q, \sigma), e_{min}), \upsilon)$
    if $\alpha(q) > c$ is even and greater than c, and
  - $\delta'' : ((q, c), \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon}\Big(((\delta(q, \sigma), c), \upsilon) \wedge \bigwedge_{\upsilon \neq \upsilon' \in \Upsilon}((\delta(q, \sigma), e_{min}), \upsilon)\Big)$
    if $\alpha(q) < c$ is even and smaller than c.
- The coloring function α'' maps the states $Q \times \{e_{min}\}$ to 2 and the remaining states to 1.

The states in Q reflect the first phase of the acceptance game on $\mathcal{G}_\mathcal{D}^\mathcal{T}$: Player accept moves to a position of her choice ($\bigvee_{\upsilon \in \Upsilon}(\delta(q, \sigma), \upsilon)$) and eventually moves on to the second phase ($\delta''(q, e_{min}), \sigma)$). The color 1 for the states in Q reflect the winning condition on infinite plays (player *accept* looses if she stays for ever in the first phase).

In the second phase, the situation is more involved, since rather than guessing the action of player *accept*, the automaton needs to cover all possible actions of player *reject*. Intuitively, the option of player *reject* to stay in the second phase is covered by sending, from a state (q, c), a copy (q', e_{min}) (with $q' = \delta(q, \sigma)$) to each direction. Since player *reject* looses when staying in the second phase indefinitely, the color of these states is 2. Additionally, if $\alpha(q)$ is odd, player *reject* could move to the third phase, which could be reflected by sending a

copy $(q, \alpha(q))$ to some direction ($\alpha(q)$ denotes the color to be stored). At the same time, we must consider the possibility that the game is in the third phase. If $\alpha(q)$ is even and greater than c, then player *accept* wins immediately (no successor send), otherwise (q', c) is sent to some successor. Since player *accept* loses by staying in the third phase indefinitely, the color of a state (q, c) with $c \neq e_{min}$ is 1. Since the situation of player *reject* becomes strictly better when the stored color c increases, we can, instead of sending (q', c) *and* (q', c') into the same direction, send only $(q', \max\{c, c'\})$. This results in the *nondeterministic* automaton ${\mathcal{O}_{\mathcal{D}}}'$.

The nondeterministic Büchi tree automaton ${\mathcal{A}_{\mathcal{D}}}' = (\Sigma, \Upsilon, Q', q_0', \delta', \alpha')$ for testing almost-sure acceptance is defined as follows:

- The set of states is set to $Q' = Q \times \mathbb{B} \times C_e^+ \cup \{\bot\}$ with initial state $q_0' = (q_0, \mathit{true}, e_{max})$, where C_e^+ denotes the set of even colors of $\mathcal{D}$, plus, if the highest color of $\mathcal{D}$ is an odd number o_{max}, $o_{max} + 1$. e_{max} denotes the highest number in C_e^+.
- The transition function is defined by:
  - $\delta' : ((q, *, c), \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon} \Big(((\delta(q, \sigma), \mathit{true}, c), \upsilon) \wedge \bigwedge_{\upsilon \neq \upsilon' \in \Upsilon} ((\delta(q, \sigma), \mathit{false}, c), \upsilon) \Big) \vee \bigwedge_{\upsilon \in \Upsilon} (\delta(q, \sigma), \mathit{false}, \min\{c, \alpha(q)\}), \upsilon)$ if $\alpha(q)$ is even,
  - $\delta' : ((q, *, c), \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon} (\bot, \upsilon)$ if $\alpha(q) > c$ is odd and greater than c,
  - $\delta' : ((q, *, c), \sigma) \mapsto \bigvee_{\upsilon \in \Upsilon} \Big(((\delta(q, \sigma), \mathit{true}, c), \upsilon) \wedge \bigwedge_{\upsilon \neq \upsilon' \in \Upsilon} ((\delta(q, \sigma), \mathit{false}, c), \upsilon) \Big)$ otherwise, and
  - $\delta' : (\bot, \sigma) \mapsto \bigwedge_{\upsilon \in \Upsilon} (\bot, \upsilon)$.
- The coloring function α' maps $Q \times \{\mathit{true}\} \times C_e^+$ and the error state $\bot$ to 1 and $Q \times \{\mathit{false}\} \times C_e^+$ to 2.

In almost-sure acceptance, the situation is slightly more involved. The states keep three pieces of information: The state of the deterministic word automaton, the information, if the game *could* be in the second phase, and a color, which reflects that the third phase could have been entered from a state in this color. The color is initialized to e_{max}, which is greater than all odd colors. From every point of the computation tree, one or no successor can refer to the second phase: No successor, if player *accept* would move to the third

phase, and one successor otherwise. Player *accept* loses if and only if there is a trace where he eventually stays indefinitely in the second phase, or if there is a trace where he eventually moves to the third phase in a state $(q, *, *)$ and then reaches a state $(q', *, *)$ with odd color $\alpha(q') > \alpha(q)$. The latter is modeled by moving to the designated error state $\bot$. The remaining information can be handled by storing the (even) color $\alpha(q)$ every time player *accept* would move to the third phase $(\bigwedge_{\upsilon\in\Upsilon}((\delta(q,\sigma), \mathit{false}, \min\{c, \alpha(q)\}), \upsilon))$ *or* by marking the direction player accept would choose when staying in the second phase $(\bigvee_{\upsilon\in\Upsilon}((\delta(q,\sigma), \mathit{true}, c), \upsilon) \wedge \bigwedge_{\upsilon\neq\upsilon'\in\Upsilon}((\delta(q,\sigma), \mathit{false}, c), \upsilon))$.

Obviously, a transition-system is rejected by $\mathcal{A}_{\mathcal{G}}{}'$ if and only if the acceptance game on $\mathcal{G}_{\mathcal{D}}^{\mathcal{T}}$ is won by player *reject*. □

These automata additionally have the pleasant property that their transition tables are short (at most $|\Upsilon| + 1$ entries for each state/input-letter pair).

The step to input-preserving transition-systems is a small one. The respective automaton can be multiplied with a deterministic safety automaton that checks if the label always agrees with the direction. The small transition table property is preserved by this transformation.

Theorem 8.5 *Given an alternating tree automaton $\mathcal{A}$ over $\Upsilon \times \Sigma$-labeled Υ-transition-systems, we can construct an alternating tree automaton $\mathcal{A}'$ over $\Upsilon \times \Sigma$-labeled Υ-transition-systems that accepts a transition-system $\mathcal{T}$ if and only if it is input-preserving and accepted by $\mathcal{A}$. If $\mathcal{A}$ has n states, $\mathcal{A}'$ has at most $n \cdot |\Upsilon|$ states. If $\mathcal{A}$ is a nondeterministic, universal, weak, Büchi, or Co-Büchi automaton, so is $\mathcal{A}'$.*

Proof: The deterministic safety automaton $\mathcal{S} = (\Upsilon \times \Sigma, \Upsilon, \Upsilon, \upsilon_0, \delta_s)$ with transition function

- $\delta : \big(q, (\upsilon, \sigma)\big) \mapsto \bigwedge_{\upsilon'\in\Upsilon}(\upsilon', \upsilon')$ if $q = \upsilon$, and
- $\delta : \big(q, (\upsilon, \sigma)\big) \mapsto \mathit{false}$) otherwise

accepts a transition system if and only if it is input preserving.

To construct $\mathcal{A}'$ it suffices to intersect $\mathcal{A}$ and $\mathcal{S}$, resulting in an automaton with $n \cdot |\upsilon|$ states for (non)deterministic automata (product construction), and in an automaton with $n + |\Upsilon| + 1$ states for alternating automata (by adding a fresh initial state q_0' with $\delta'\big(q_0', (\upsilon, \sigma)\big) \mapsto \delta\big(q_0, (\upsilon, \sigma)\big) \wedge \delta_s\big(\upsilon_0, (\upsilon, \sigma)\big)$, and reusing the transition functions δ and δ' on their respective co-domain). □

8.4 Temporal Logics

The basic techniques for trace languages and ε-environments provided in Section 8.3 are transferred to temporal logics in this section. For the linear-time temporal logic LTL, the techniques from the previous section can easily be applied: It suffices to translate an LTL formula into an equivalent deterministic word automaton, and then use the results of Section 8.3.

For probabilistic systems, the almost-sure/observable semantics for LTL inspire a redefinition of CTL* semantics [HJ94]: Universal path quantification ($A\pi$) can be interpreted as the property that the probability measure of the paths satisfying π is 1, and existential path quantification can be interpreted as the property that the probability measure of the paths satisfying π is greater than 0.

Liner-Time Logic. Converting LTL formulas to deterministic word automata is well established. Combining the Theorem 7.1 and 7.3 directly implies:

Theorem 8.6 *Given an LTL specification φ, we can construct a deterministic word automaton $\mathcal{D}_\varphi$ that accepts exactly the models of φ. The number of states of $\mathcal{D}_\varphi$ is doubly exponential in the length of φ.* □

Given an LTL specification φ, we can, by the Theorems 8.6, 8.4 and 8.5, construct a nondeterministic Büchi tree automaton $\mathcal{N}_\varphi$ that accepts an input-preserving $2^I \times 2^O$-labeled 2^I-transition-system if and only if it almost-surely (observably) satisfies φ, such that the number of states of $\mathcal{N}_\varphi$ is doubly exponential in the length of φ. A constructive non-emptiness test for $\mathcal{N}_\varphi$ can be performed in time quadratic in the number of states of $\mathcal{N}_\varphi$ (Theorem 4.7).

Corollary 8.7 *Given an LTL specification φ we can, in time doubly-exponential in the length of φ, construct an input-preserving $2^I \times 2^O$-labeled 2^I-transition-system that almost-surely (observably) satisfies φ, or show that no such transition-system exists, in time doubly-exponential in the length of φ.* □

It turns out that this upper bound is sharp.

Theorem 8.8 *The LTL synthesis problem is 2EXPTIME-complete.*

Proof: The upper bound is established by Corollary 8.7. To establish a matching lower bound, consider the ω-regular trace language

$$\mathcal{L}_n \;=\; \{\, \{0,1,2,3\}^* \cdot 3 \cdot \{0,1,2\}^* \cdot 2 \cdot v \cdot 2 \cdot \{0,1,2\}^* \cdot 3 \cdot v \cdot \{0,1,2\}^\omega \mid v \in \{0,1\}^n \}.$$

While $\mathcal{L}_n$ can be expressed by an LTL formula with size quadratic in n, any automaton accepting $\mathcal{L}_n$ necessarily has at least 2^n states [KV95] (since it must continuously update the set of *subsets* of $\{0,1\}$ words of length n that have occurred between two 2 symbols since the last 3).

Consider a system with two Boolean input variables i_1 and i_2, and a single output variable o. One can use i_1 and i_2 to encode the letters $0,1,2,3$, and represent the language $\mathcal{L}_n$ by a formula φ_n (of length quadratic in n).

The specification $\psi_n = \varphi_n \leftrightarrow FGo$ can only be satisfied by a transition-system with at least $O(2^n)$ states, regardless if in classical, almost-sure or observable semantics, since the transition-system *always* needs to react on an additional 3 (for example, by setting the value of the output variable to *true* or *false* n steps after a 3 was read and keeping it constant otherwise). □

Branching-Time. In the branching-time case, one can use the fact that $E\psi$ and $A\psi$ are state-formulas. We call the strict subformulas of a CTL* specification φ of this special form the *basic* subformulas of φ, denoted $basic(\varphi)$. Testing if a transition-system $\mathcal{T}$ satisfies a CTL* formula φ can be reduced to testing if the labels of $\mathcal{T}$ can be extended with suitable truth values for the basic subformulas of φ. The correct labels can be guessed on the fly.

Theorem 8.9 *Given a CTL* specification φ we can construct a weak alternating tree automaton $\mathcal{A}$ which accepts an $2^I \times 2^O$-labeled 2^I-transition-system if and only if it satisfies φ. The number of states of $\mathcal{A}$ is doubly-exponential in the length of φ.*

Proof: In our construction, the values of the basic formulas are guessed. Let $\mathcal{A}_\psi = (\Sigma^\psi, 2^I, Q^\psi, q_0^\psi, \delta^\psi, \alpha^\psi)$ denote the weak alternating tree automaton that accepts the models of a basic formula $\psi = E\psi'$ or $\psi = A\psi'$ of φ (or of φ itself), where the basic subformulas of ψ are provided as atomic propositions. $\mathcal{A}_\psi$ can be constructed by the method introduced in Theorem 8.3. The number of states of $\mathcal{A}_\psi$ is doubly exponential in the number of states of ψ. Let $\mathcal{A}_{\overline{\psi}} = (\Sigma^\psi, 2^I, Q^{\overline{\psi}}, q_0^{\overline{\psi}}, \delta^{\overline{\psi}}, \alpha^{\overline{\psi}})$ denote the weak alternating automaton dual to $\mathcal{A}_\psi$.

We assume without loss of generality that φ is basic (otherwise we can replace the state formula φ by $A\varphi$ or $E\varphi$ without changing the semantics) and define the weak alternating tree automaton $\mathcal{A} = (2^I \times 2^O, , 2^I, Q, q_0, \delta, \alpha)$ as follows: The states $Q = Q^\varphi \cup \bigcup_{\psi \in basic(\varphi)} (Q^\psi \cup Q^{\overline{\psi}})$ are formed by the states of the single weak alternating automata $\mathcal{A}_\psi$, and the initial state $q_0 = q_0^\varphi$ is the initial state

of $\mathcal{A}_\varphi$. The transition function is defined such that

$$\delta(q^\psi, \sigma) = \bigvee_{\Psi \subseteq basic(\psi)} \Big(\delta^\psi(q^\psi, \sigma \cup \Psi) \wedge \bigwedge_{\psi' \in \Psi} \delta(q_0^{\psi'}, \sigma) \wedge \bigwedge_{\psi' \in basic(\psi) \setminus \Psi} \delta(q_0^{\overline{\psi'}}, \sigma) \Big)$$

holds true. The coloring function maps a state q^ψ with even (odd) color $\alpha^\psi(q^\psi)$ in $\mathcal{A}_\psi$ to an even (odd) color, such that the weakness criterion is preserved.

Intuitively, the truth of the single basic subformulas is guessed on the fly. To demonstrate that guessing these values is safe, we show that player *accept* has a winning strategy in the acceptance game if and only if he as a winning strategy where he always guesses the validity of all basic subformulas correctly. This can be demonstrated by induction along the structure of φ: Assume that player *accept* has a winning strategy where the truth value of some subformula is guessed incorrectly. Then there is a basic subformula ψ whose truth value is eventually guessed *incorrectly*, but the truth values of the basic subformulas of ψ are always guessed correctly. Then, for a state s in the transition-system $\mathcal{T}$ where the truth of ψ was eventually guessed incorrectly (without loss of generality to *true*), player accept has a winning strategy from (q_0^ψ, s) in the acceptance game, such that all values of basic subformulas of ψ are guessed correctly. Then player *accept* has a winning strategy in $\mathcal{A}_\psi$ when the labeling of $\mathcal{T}$ are enriched by the correct values for the basic subformulas of ψ (the winning strategy is the winning strategy from $\mathcal{A}$, with the simplification that the correct values need not be guessed). But in this case ψ *is* valid in s. $\lightning$ □

The automaton $\mathcal{A}_\varphi$ constructed by Theorem 8.9 can, by Corollary 4.6, be turned into an equivalent nondeterministic Büchi tree automaton $\mathcal{N}_\varphi$ with exponentially more states than $\mathcal{A}_\varphi$. The language of $\mathcal{N}_\varphi$ can be restricted to input-preserving transition-systems (Theorem 8.5). A transition-system accepted by $\mathcal{A}_\varphi$ can be constructed by solving the emptiness game for the resulting automaton (Theorem 4.7).

Corollary 8.10 *Given a CTL* specification φ we can construct an input-preserving $2^I \times 2^O$-labeled 2^I-transition-system, or proof that no such system exists, in time triply exponential in the length of φ.*

Theorem 8.8 provides a 2EXPTIME lower bound, which leaves the exact characterization of the complexity of the CTL* synthesis problem open. For its important sub-logic CTL, the complexity coincides with the synthesis complexity for classical semantics.

Theorem 8.11 *The CTL synthesis problem is EXPTIME-complete.*

Proof: In CTL, each path quantifier refers to a path formula of the form $\psi_1 U \psi_2$, $G\psi_1$, or $X\psi_1$, where ψ_1 and ψ_2 are propositional (when basic formulas are viewed as propositions). For such path formulas (and their negations) acceptance of a path can be tested by a deterministic word automaton with three, two, or three states, respectively. The alternating automaton constructed by Theorem 8.9 is therefore only *linear* in the length of the specification, and emptiness can be checked in time exponential in the length of the specification by first nondeterminizing this automaton (Corollary 4.6) and then performing a constructive non-emptiness test for the resulting automaton (Theorem 4.7).

To demonstrate EXPTIME-hardness, we reduce solving the two player game PEEK-G_4 [SC79] to CTL synthesis. An instance of this game is a four-tuple $\langle X, Y, Z, \varphi \rangle$, where X and Y are disjoint sets of Boolean variables with the intuition that X is under the control of the system and Y is under the control of the environment. $Z \subseteq X \cup Y$ denotes the variables which initially hold true and φ is a propositional formula over the variables $X \cup Y$. The game is played in rounds where first the system can change the value of at most one variable in X, followed by a decision of the environment to change the value of at most one variable in Y. The system wins the game if and only if φ is eventually satisfied (after the move of the system). To determine the winner of such games is EXPTIME-hard [SC79].

An instance of this game can be reduced to the synthesis problem for a system with one input-variable i, two output variables o_1 and o_2, and a CTL specification ψ quadratic in $|X|+|Y|$ and linear in φ. $\psi = \psi_0 \wedge \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_\varphi$ is a conjunction of the following five CTL formulas:

- ψ_0 requires that the first $|X|$ values of o_1 reflect (on every path) the initial truth value of the variables in X (defined by $X \cap Z$) and the following $|Y|$ values of o_1 reflect the initial truth value of the variables in Y.
- ψ_1 requires that o_2 is *true* exactly every $|X| + |Y|$ steps (and initially) on every path.
- ψ_2 requires that at most one value of the variables o_1 within $|X| - 1$ steps after o_2 was last set to *true* (including the current step) differs from the value of o_1 $|X| + |Y|$ steps earlier.
- ψ_3 states that within $|X|$ to $|X| + |Y| - 1$ steps after o_2 was set true, the value of the variable o_1 is different from its value $|X| + |Y|$ steps earlier if and only if
  - the value of the input variable is *true*, and

  - the values of the previous input variables since $|X|$ steps after o_2 was last set to *true* were all *false*.

- ψ_φ requires that, for all paths, there is eventually a position where o_2 is *true* and along the path where i is *false* for the following $|X| + |Y|$ steps, the following $|X| + |Y|$ values of o_1 (including the current value) satisfy φ (when interpreted as predicates for φ).

ψ_2 and ψ_3 refer to the changing of at most one assignment for the variables of X and Y by the system and the environment, respectively, ψ_0 initializes the game and ψ_1 guarantees that o_2 can be used as a flag, indicating that a round starts. ψ_φ reflects the winning condition of the game. An input-preserving transition-system that satisfies ψ (in classical semantics as well as in almost-sure/observable semantics) defines a winning strategy for $\langle X, Y, Z, \varphi\rangle$ and vice versa. □

8.5 0-Environments

0-environments can "emphasize" each single path by assigning a probability measure of 1 (if the probability of each single action can be chosen from $[0, 1]$) or arbitrarily close to 1 (if the probability of each single action can be chosen from $]0, 1]$). For the latter case, consider an assignment of the probability $1-2^i \cdot \varepsilon$ for staying on the path desired by the environment in the i-th step for some $\varepsilon > 0$[1].

Consequently, the LTL synthesis problem coincides for almost-sure and observable semantics with the LTL synthesis problem for classical semantics, which is 2EXPTIME-complete [PR89a].

For almost-sure/observable CTL* semantics this implies that existential and universal path quantifiers coincide. Consequently, a transition-system $\mathcal{T}$ is a model of a CTL* specification φ if and only if $\mathcal{T}$ is a model of a specification φ' in classical semantics, where φ' is obtained from φ by replacing all existential path quantifiers by universal path quantifiers. This implies EXPTIME and 2EXPTIME upper bounds for the CTL and CTL* synthesis problem [KV99], respectively.

On the other hand, in classical semantics each specification ψ can be translated to an equivalent specification ψ' by replacing each occurrence of an existential path quantifier E by the sequence $\neg A \neg$. Since the length of ψ' is linear

[1] The probability measure of the path is, in this case, greater than $1 - \varepsilon$, and can therefore be chosen arbitrarily close to 1 by the 0-environment.

in the length of ψ and the classical semantics for ψ' coincides with the almost-sure/observable semantics, the matching lower bounds for the CTL and CTL* synthesis problem [KV99] are preserved as well.

8.6 Conclusions

In this chapter, constructive decision procedures for the LTL, CTL and CTL* synthesis problems under the assumption of 0-environments and ε-environments have been introduced. While the semantics for 0-environments essentially reflect the classical semantics and practically all established results trivially carry over, the results for ε-environments provide interesting new insights.

The results of this chapter show that the complexity of synthesizing transition-systems that satisfy an LTL or CTL specification φ in almost-sure/observable semantics is, under the assumption of ε-environments, equivalent to the complexity in classical semantics. While the complexity coincides, the language classes for LTL are at the same time simpler and more involved than for classical semantics: They are simpler in the sense that the languages are recognizable by *weak* alternating automata, and more involved in the sense that they cannot be recognized by deterministic automata.

Two interesting questions deserve further study: The exact complexity of CTL* synthesis in almost-sure/observable semantics, and the influence of incomplete information on the complexity of the LTL[2] synthesis problem. These problems may be closely interrelated: In classical semantics, both problems can be solved through the existence of alternating automata that are only exponential in the length of a CTL* formula φ, which recognizes the models of φ. It does not seem unlikely that similar solutions exist for almost-sure/observable semantics, taking into account that model-checking remains PSPACE-complete (Courcoubetis and Yannakakis PSPACE result for LTL model-checking [CY95] trivially extends to CTL*).

An interesting side effect of using an automata-based synthesis algorithm is the possibility to extend the results for single-process synthesis directly to distributed synthesis.

[2]For CTL and CTL* synthesis, incomplete information can be handled using established automata-based techniques [KV97b] (cf. Lemma 6.7).

Chapter 9

Semi-Automatic Synthesis

Abstract

In this chapter, a sound and complete compositional proof rule for distributed synthesis is proposed. Applying this proof rule only requires the manual strengthening of the specification into a conjunction of formulas that can be guaranteed by individual black-box processes. All premises of the proof rule can be checked automatically.

For this purpose, we give an automata-theoretic synthesis algorithm for single processes in distributed architectures. Different to the setting of distributed synthesis, the local environment of a process is unknown in the process of synthesis and must be assumed to be hostile. We therefore consider reactive environments that have the power to disable some of their own actions, and provide methods for synthesis (and realizability checking) in this setting. We establish upper bounds for CTL (2EXPTIME) and CTL* (3EXPTIME) synthesis with incomplete information, matching the known lower bounds for these problems, and provide matching upper and lower bounds for μ-calculus synthesis (2EXPTIME) with complete or incomplete information. Synthesis in reactive environments is harder than synthesis in maximal environments, where CTL, CTL* and μ-calculus synthesis are EXPTIME-complete, 2EXPTIME-complete and EXPTIME-complete, respectively.

9.1 Introduction

In the synthesis of distributed systems, we transform a given specification into a collection of finite-state programs that are guaranteed to satisfy the specification when combined according to a given architecture. In the Chapter 6, we have seen that distributed synthesis can be done automatically for fork-free architectures. However, as soon as the architecture contains an *information fork*, the problem becomes undecidable.

In this chapter, we investigate a *semi-automatic* approach, where we synthesize one process at a time. It turns out that the synthesis of a single process can be done automatically and it is always possible to decompose a realizable specification into a conjunction of properties that can be guaranteed by single processes. This approach therefore works for all distributed architectures, including those with information forks.

The main difference between the classic synthesis problems and the problem studied in this chapter is that the environment of the process now consists of multiple constituents. In addition to the external environment, the process may interact with the white-box processes and with the other black-box processes in the system. Both the behavior of the external environment and the behavior of the white-box processes are known *a priori*: In our setting, we assume that the behavior of the external environment is maximal and that the behavior of each white-box process is given as a (possibly nondeterministic) finite-state automaton. (Technically, the environment is treated as an ordinary white-box process.) By contrast, the strategies of the other black-box processes are unknown. From the point of view of the considered process, their behavior therefore appears reactive: At any point, they may disable some (but not all) of their possible responses.

We call a process implementation *resilient* if the specification is satisfied *independently* of how the other black-box processes are implemented. (And even if they get access to the complete system state.) We demonstrate that the resilient synthesis problem is 2EXPTIME-complete for CTL and μ-calculus specifications and 3EXPTIME-complete for specifications in CTL*. Our proof is constructive: We introduce an automata-theoretic algorithm that determines for a temporal specification and a process in a distributed architecture whether there exists a resilient implementation and, if yes, computes one such implementation. We establish 2EXPTIME and 3EXPTIME upper bounds for synthesis with incomplete information in case of μ-calculus and CTL* specifications, respectively. These upper bounds match the lower bounds for checking resilient realizability

for CTL and CTL* specifications, respectively, under the assumption of complete information and a monolithic environment [KMTV00].

We propose to use the new synthesis algorithm in a compositional synthesis rule for distributed synthesis. Applying our synthesis rule requires the strengthening of the specification into a conjunction of formulas for which resilient implementations exist. The rule is complete: If a specification can be implemented, then there also exists a strengthening for which that implementation is resilient. Since the synthesis of resilient implementations is automatic, the strengthening is the only manual step in the application of the rule.

The remainder of the chapter is structured as follows. In Section 9.2, we formally introduce the synthesis problem studied in this chapter. We introduce the compositional synthesis rule in Section 9.3 and illustrate its application on a simple example in Section 9.4. In Section 9.5, we prove the completeness of the rule. The synthesis algorithm is presented in Section 9.6.

9.2 Resilient Realizability

A set of strategies $\{p_b : (2^{I_b})^* \to \mathcal{O}_b\}_{b \in Q}$ of a set $Q \subseteq B$ of black-box processes is called a *resilient realization* of a specification φ if the computation tree satisfies φ independently of the other black-box processes: That is, for every $\mathcal{O}_{B \setminus Q}$-labeled 2^{Π}-tree $\langle (2^{\Pi})^*, p_{B \setminus Q} \rangle$, representing the behavior of the black-box processes in $B \setminus Q$, the computation tree $\langle \| p \|, dir \rangle$ with $p = p_Q \oplus p_{B \setminus Q} \oplus p_W$ is a model of φ.

A specification φ is *resiliently realizable* by a set $Q \subseteq B$ of black-box processes, denoted by $(\mathbf{A}, Q) \vDash \varphi$, if there exists a set of finite-state strategies for the processes in Q that are a resilient realization of φ:

$$(\mathbf{A}, Q) \vDash \varphi :\Leftrightarrow \exists \{p_b : (2^{I_b})^* \to \mathcal{O}_b \mid b \in Q\}. \forall \{p_{B \setminus Q} : (2^{\Pi})^* \to \mathcal{O}_{B \setminus Q}. \langle \| \bigoplus_{b \in Q} p_b \oplus p_{B \setminus Q} \oplus p_W \|, dir \rangle \vDash \varphi$$

For the complete set of black-box processes ($Q = B$) realizability and resilient realizability coincide: A specification φ is *realizable* if it is (resiliently) realizable by the entire set B of black-box processes.

9.3 The Compositional Synthesis Rule

Building on the definition of resilient realizability, we now introduce a semi-automatic approach to distributed synthesis. We define a compositional synthe-

sis rule that establishes the realizability of a specification by showing that the specification can be strengthened into a conjunction of local specifications for the individual processes, such that each local specification is resiliently realized by its process. While the strengthening of the specification must be done manually, we will show in Section 9.6 that all premises of the synthesis rule can be checked automatically.

For a distributed architecture $\mathbf{A}$ with a set of black-box processes $B = \{b_1, \cdots, b_n\}$, and CTL* or μ-calculus formulas $\psi, \varphi_{b_1}, \ldots \varphi_{b_n}$,

$$\begin{array}{lll} \text{(R1)} & (\mathbf{A}, \{b_1\}) \models & \varphi_{b_1} \\ \vdots & \vdots & \\ \text{(R}n\text{)} & (\mathbf{A}, \{b_n\}) \models & \varphi_{b_n} \\ \text{(S)} & (\mathbf{A}, \emptyset) \models & \bigwedge\limits_{b \in B} \varphi_b \rightarrow \psi \\ \hline & (\mathbf{A}, B) \models & \psi \end{array}$$

Theorem 9.1 *The compositional synthesis rule is sound.*

Proof: Premises (R1) through (Rn) prove that each local specification φ_{b_i} is resiliently realized by the respective black-box process b_i:

$$(\mathbf{A}, \{b_i\}) \models \varphi_{b_i} \Leftrightarrow \exists p_{b_i} : (2^{I_{b_i}})^* \rightarrow \mathcal{O}_{b_i}. \forall p_{B \setminus \{b_i\}} : (2^V)^* \rightarrow \mathcal{O}_{B \setminus \{b_i\}}. \\ \langle \| p_{b_i} \oplus p_{B \setminus \{b_i\}} \oplus p_W \|, dir \rangle \models \varphi_{b_i}.$$

Consequently, such strategies p_{b_i} can be fixed independently. The resulting implementation $\{p_{b_i}\}_{b_i \in B}$ satisfies φ_{b_i} for all $b_i \in B$. Hence, $(\mathbf{A}, B) \models \bigwedge_{b_i \in B} \varphi_{b_i}$ holds true:

$$\exists \{p_{b_i} : (2^{I_{b_i}})^* \rightarrow \mathcal{O}_{b_i} \mid b_i \in B\}. \langle \| \bigoplus_{b_i \in B} p_{b_i} \oplus p_W \|, dir \rangle \models \bigwedge_{b_i \in B} \varphi_{b_i}.$$

Premise (S) shows that the conjunction of the local specifications $\bigwedge_{b_i \in B} \varphi_{b_i}$ implies the system specification ψ. Therefore, every implementation that satisfies all local specifications must also satisfy ψ:

$$
\begin{aligned}
&(\mathbf{A}, \emptyset) \vDash \bigwedge_{b_i \in B} \varphi_{b_i} \rightarrow \psi \\
\Leftrightarrow\ &\forall p_B{:}(2^V)^* \rightarrow \mathcal{O}_B.\langle \| p_B \oplus p_W \|, dir\rangle \vDash \bigwedge_{b_i \in B} \varphi_{b_i} \rightarrow \psi \\
\Leftrightarrow\ &\forall p_B : (2^V)^* \rightarrow \mathcal{O}_B.\langle \| p_B \oplus p_W \|, dir\rangle \vDash \bigwedge_{b_i \in B} \varphi_{b_i} \Rightarrow \langle \| p_B \oplus p_W \|, dir\rangle \vDash \psi.
\end{aligned}
$$

In particular, the computation tree defined by $\{p_{b_i}\}_{b_i \in B}$ must satisfy ψ. Hence, the compositional synthesis rule is sound.

9.4 Example

We illustrate the compositional synthesis rule with a simple distributed *shared-resource* application. The system architecture is shown in Figure 9.1a. The external environment *env* is depicted as a circle, the two black-box processes p_1 and p_2 as filled rectangles, and the white-box "Arbiter" process as an empty rectangle.

env can request *access* to the resource by setting the *request* variable of one of the two black-box processes p_1 and p_2. The unconstrained external environment is modeled as a white-box process that shows every possible behavior, that is, p_{env} is the constant function that maps every input history to $2^{\{request_1, request_2\}}$. The white-box process Arbiter, whose deterministic implementation is shown in Figure 9.1b, ensures mutual exclusion by passing a *grant* back and forth between p_1 and p_2, such that each process retains the grant until the respective *release* variable is set.

We specify the expected behavior of the shared-resource system as a conjunction $\psi = \psi_1 \wedge \psi_2 \wedge \psi_3$ of three CTL* formulas, where the first two formulas specify that there is a way for both processes to use the resource infinitely often (ψ_i = EGF $access_i$ for $i \in \{1, 2\}$) and the third formula specifies mutual exclusion (ψ_3 = AG $\neg(access_1 \wedge access_2)$).

Obviously, neither p_1 nor p_2 can guarantee ψ for *all* possible implementations of the other process (for example, if p_1 constantly sets its $access_1$ variable to *true*, p_2 cannot avoid violating mutual exclusion if it is to obtain access along some branch).

Using the proof rule, we need to strengthen ψ into two separate properties φ_{p_1} and φ_{p_2} that can be resiliently realized by p_1 and p_2, respectively. A natural assumption to be made by process p_{3-i} about process p_i is that there is a path such that process p_i infinitely often releases the grant ($\alpha_1^{p_i}$ = EGF $release_i$) and that, on every path, p_i only accesses the resource when permitted by the arbiter ($\alpha_2^{p_i}$ = AG $access_i \rightarrow grant_i$). By adding these assumptions, we obtain

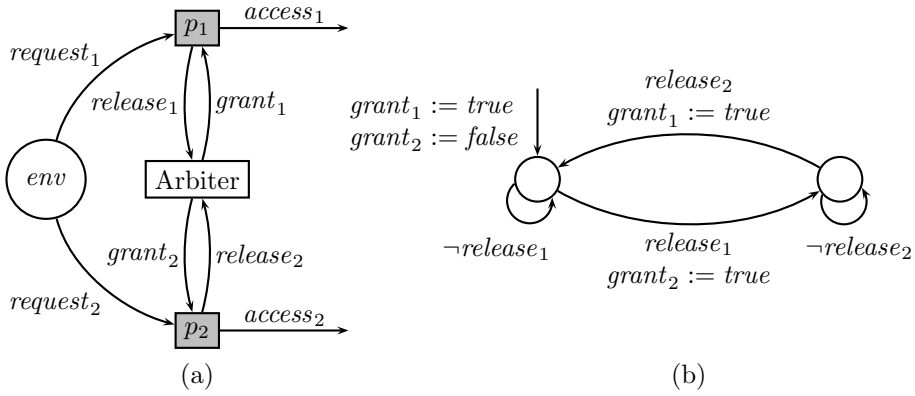


Figure 9.1: A simple distributed shared-resource application. (a) The system architecture. An edge between two process nodes p and q labeled with variable v indicates that v is an output variable of process p and an input variable of process q. (b) The implementation of the white-box process Arbiter, represented as a finite-state Mealy machine. The edges are labeled with the value of the input variables and with new assignments to the output variables (if the value of the output variables changes).

a strengthened specification $\varphi = \varphi_{p_1} \wedge \varphi_{p_2}$ where

$$\varphi_{p_i} \;=\; \alpha_1^{p_i} \wedge \alpha_2^{p_i} \quad \wedge \quad (\alpha_1^{p_{3-i}} \wedge \alpha_2^{p_{3-i}} \;\rightarrow\; \psi).$$

Once the auxiliary formulas φ_{p_1} and φ_{p_2} have been defined, a resilient implementation can be found for both processes. For example, process p_i can guarantee φ_{p_i} by setting $access_i$ after each $request_i$ as soon as $grant_i$ becomes *true* and by setting $release_i$ in the immediately following state.

In terms of the traditional system development process, the strengthening in our proof rule can be understood as the definition of an abstract interface or contract between the processes. Typically, the user can choose from multiple correct contracts. In the shared-resource application, the user might, for example, alternatively specify strict turn-taking by strengthening ψ to φ'_{p_1} and φ'_{p_2}, where φ'_{p_1} requires that p_1 accesses the resource exactly after every *odd* number of steps and φ'_{p_2} requires that p_2 accesses the resource exactly after every *even* number of steps. The resilient implementations of this strengthening are slightly unorthodox (the arbiter is ignored), but are also guaranteed to satisfy ψ.

As a final remark about this example, let us convince ourselves that it is really necessary to consider *resilient* implementations. Suppose that, in a hypothetical alternative proof rule, we require only the (cheaper) realizability in a maximal local environment. It is now possible to strengthen ψ into two formulas φ''_{p_1} and φ''_{p_2} that can, in a maximal local environment, be guaranteed by p_1 and p_2, respectively, but whose conjunction is equivalent to *false*. For example, p_1 can easily guarantee $AG\,\neg release_1$, while any implementation of p_2 guarantees $EF\,release_1$ in a maximal local environment.

9.5 Completeness

To demonstrate the completeness of the compositional synthesis rule, we show that the auxiliary formulas $\varphi_{b_1}, \ldots, \varphi_{b_n}$ required in the rule can be derived from a distributed implementation $\{p_{b_i}\}_{b_i \in B}$ that satisfies the specification ψ. Given an implementation p_b of a black-box process b, we define a CTL formula φ_b that is a *strict characterization* of the behavior of b and the white-box processes. Strict characterization means that

1. p_b is a resilient implementation of φ_b, and
2. for all other implementations $\{p'_{b_i}\}_{b_i \in B}$ that realize φ_b, the implementations p_b and p'_b have the same computation trees:

$$\forall p_{B \setminus \{b\}} : (2^V)^* \to \mathcal{O}_{B \setminus \{b\}}.\, \|p_b \oplus p_{B \setminus \{b\}} \oplus p_W \oplus p_{env}\| = \|p'_b \oplus p_{B \setminus \{b\}} \oplus p_W \oplus p_{env}\|.$$

Condition (1) guarantees that premise (Ri) of the proof rule is satisfied for each black-box process b_i, and Condition (2) guarantees that premise (S) is satisfied.

The arbiter depicted in Figure 9.1(b) has the following strict characterization:

$$\varphi_{arbiter} = \varphi_0 \wedge \mathrm{AG}(\varphi_0 \vee \varphi_1), \text{ with}$$

$$\varphi_0 = grant_1 \wedge \neg grant_2 \wedge (release_1 \to \mathrm{AX}(\neg grant_1 \wedge grant_2)) \\ \wedge (\neg release_1 \to \mathrm{AX}(grant_1 \wedge \neg grant_2)), \text{ and}$$

$$\varphi_1 = \neg grant_1 \wedge grant_2 \wedge (release_2 \to \mathrm{AX}(grant_1 \wedge \neg grant_2)) \\ \wedge (\neg release_2 \to \mathrm{AX}(\neg grant_1 \wedge grant_2)).$$

Let the implementation for the processes $b \in B$ and for the white-box processes be given as finite-state transducers, which we combine into a single transducer T_B with a set S_B of states and an initial state $s_0^B \in S_B$. Additionally, we

construct a finite-state transducer T_b with a set S_b of states and initial state s_0^b for the product of each single black-box process b and the white-box processes.

We define a CTL formula φ_b such that the models of φ_b are the trees obtained by unraveling T_b. To construct φ_b, we give a formula φ_s for each state $s \in S_b$ which ensures that, for the next $max\{|S_B|, |S_b|\} + 1$ steps, the tree corresponds to the unraveling of T starting in state s. Since the other black-box processes are unknown, φ_s does not require that all branches of the unraveling exist, but rather that, provided they do exist, the reaction is in accordance with T_b. The specification $\varphi_b = \varphi_{s_0^b} \wedge \mathrm{AG} \bigvee_{s \in S_b} \varphi_s$ requires that φ_{s_0} holds initially, and that always some φ_s holds true.

The formula φ_b is a strict characterization of the behavior of b and the white-box processes. As required by Condition (1), p_b is a resilient implementation of φ_b. For Condition (2), note that resilient realization includes realization in a maximal environment as a special case: Hence, $p'_b \oplus \bigoplus_{w \in W} p_w$ must react to any input from the other black-box processes exactly like $p_b \oplus \bigoplus_{w \in W} p_w$.

An unraveling of height $|S_b| + 1$ suffices for strict characterizations, and an unraveling of height $|S_B| + 1$ suffices to guarantee that the formula $\bigwedge_{b \in B} \varphi_B$ is a strict characterization of the overall system.

Theorem 9.2 *The compositional synthesis rule is complete.*

Proof: Assume that ψ is realizable and let $\{p_{b_i} : (2^{I_{b_i}})^* \to \mathcal{O}_{b_i}\}_{b_i \in B}$ be a realization of ψ. For each black-box process b_i, we can infer a formula φ_{b_i}, which is a strict characterization of the behavior of b_i and the white-box processes, that is, of $p_{b_i} \oplus p_W$. Premises (Ri) of the compositional synthesis rule are satisfied because for each φ_{b_i}, the given p_{b_i} is a resilient realization.

The conjunction of the single strict characterizations define a strict specification of the overall system. Consequently, each implementation that satisfies $\bigwedge_{b_i \in B} \varphi_{b_i}$ also satisfies ψ, and Premise (S) holds true. □

9.6 Synthesis of Resilient Implementations

We now develop a procedure that checks if a specification is resiliently realizable by a single black-process b, $(\mathbf{A}, \{b\}) \vDash \varphi$, as required for Premises (R1) through (Rn), and a procedure that checks if a specification is resiliently realized by the empty set of black-box processes, $(\mathbf{A}, \emptyset) \vDash \varphi$, as required for Premise (S).

9.6.1 Overview

The synthesis algorithm is a generalization of the synthesis algorithm for 1-black-box architectures (Subsection 6.5.1). We consider the resilient realizability problem $(\mathbf{A}, \{b\}) \vDash \varphi$ for an architecture with a black-box process b. Given such an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$, an AT$\mu$C specification φ, and a set $P_W = \{p_w\}_{w \in W}$ of implementations for the white-box processes, the following algorithm constructs a nondeterministic automaton $\mathcal{G}$, which accepts an implementation p_b of the black-box process b if and only if it resiliently realizes φ, that is, if

$$\forall p_{B \setminus Q} : (2^\Pi)^* \to \mathcal{O}_{B \setminus Q}. \langle \| p_b \oplus p_{B \setminus Q} \oplus p_W \|, \mathit{dir} \rangle \vDash \varphi.$$

Realizability can then be checked by solving the emptiness game for $\mathcal{F}$. The synthesis algorithm uses the following automata operations:

- **From specification to automata.** First, a specification φ is turned into an ε-free ACG $\mathcal{A}$ that accepts exactly the models of φ (Theorem 4.1 and Lemma 4.2).
- **From models to implementations.** We then transform $\mathcal{A}$ into an alternating tree automaton $\mathcal{B}$ that accepts a relaxed implementation $\langle (2^\Pi)^*, l \times \bigoplus_{a \in A} p'_a \rangle$ (extended by atomic propositions; $l : (2^\Pi)^* \to 2^\Pi$) with input Π if and only if $\langle \| p'_A \|, l \rangle$ is a model of φ (Lemma 6.4).
- **Quantification over black-box implementations.** In a third step, we construct an alternating automaton $\mathcal{C}$ that accepts a $2^\Pi \times \mathcal{O}_b \times \mathcal{O}_W$-labeled 2^Π-tree $\langle (2^\Pi)^*, l \times p'_b \oplus p'_W \rangle$ if and only if, for *all* $p_{B \setminus \{b\}} : (2^\Pi)^* \to \mathcal{O}_{B \setminus \{b\}}$, the $2^\Pi \times \mathcal{O}_b \times \mathcal{O}_W \times \mathcal{O}_{B \setminus b}$-labeled 2^Π-trees $\langle (2^\Pi)^*, l \times p'_b \oplus p'_W \oplus p_{B \setminus \{b\}} \rangle$ is accepted by $\mathcal{B}$ (Corollary 9.4).
- **Adjusting for white-box processes.** In the following step, we construct an alternating automaton $\mathcal{D}$ that accepts a $2^\Pi \times \mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, l \times p_b \rangle$ if and only if the $2^\Pi \times \mathcal{O}_b \oplus \mathcal{O}_W$-labeled 2^Π-tree $\langle (2^\Pi)^*, l \times p_b \oplus \bigoplus_{w \in W} p_w \rangle$ obtained by composing p_b with the implementations $P_W = \{p_w\}_{w \in W}$ of the white-box processes is accepted by $\mathcal{C}$ (Lemma 6.5).
- **Pruning the directions from the label.** We then construct an alternating automaton $\mathcal{E}$ that accepts an $\mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, p_b \rangle$ if and only if the $2^\Pi \times \mathcal{O}_b$-labeled 2^Π-tree $\langle (2^\Pi)^*, \mathit{dir} \times p_b \rangle$ obtained by adding the directions to the label is accepted by $\mathcal{D}$ (Lemma 6.6).

- **Incomplete information.** In a sixth step, we transform $\mathcal{E}$ into an alternating automaton $\mathcal{F}$ that accepts an $\mathcal{O}_b$-labeled 2^{I_b}-tree $\langle (2^{I_b})^*, p_b \rangle$ if and only if its suitable widening $\langle (2^{\Pi})^*, p_b \circ \mathit{hide}_{2^{\Pi \setminus I_b}} \rangle$ is accepted by $\mathcal{E}$ (Lemma 6.7).
- **Emptiness test.** In the last step, we test the emptiness of $\mathcal{F}$ by first constructing a nondeterministic tree automaton $\mathcal{G}$ with $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{F})$ (Corollary 4.6), and then performing a constructive non-emptiness test for $\mathcal{F}$ (Theorem 4.7).

Quantification is the only step of this synthesis algorithm that is not covered by the construction from Subsection 6.5.1.

9.6.2 Quantification

The central automata construction of this chapter covers the quantification over all possible relaxed implementations of the black-boxes $B \setminus \{b\}$. That is, we build an automaton $\mathcal{C}$ that accepts a $2^{\Pi} \times \mathcal{O}_b \otimes \mathcal{O}_W$-labeled 2^{Π}-tree if all $\mathcal{O}_{B \setminus \{b\}}$ extensions are accepted by $\mathcal{B}$:

$$\mathcal{L}(\mathcal{C}) = \{\langle (2^{\Pi})^*, l \times l' \rangle \mid l : (2^{\Pi})^* \to 2^{\Pi},\ l' : (2^{\Pi})^* \to \mathcal{O}_b \otimes \mathcal{O}_W \text{ and } \forall l'' : (2^{\Pi})^* \to \mathcal{O}_{B \setminus \{b\}}.\ \langle (2^{\Pi})^*, l \times (l' \oplus l'') \rangle \in \mathcal{L}(\mathcal{B})\}.$$

Intuitively, the automaton guesses the most hostile behavior of the remaining black-box processes. Note that for *universal* specifications the most hostile behavior of the remaining black-box processes is the *maximal* behavior, where each process $b' \in B \setminus \{b\}$ continuously enables all successors ($p_{b'}(y) = \{2^{O_{b'}}\}\ \forall y \in (2^{\Pi})^*$). The quantification step can therefore be avoided for universal specifications, provided such a maximal behavior exists (cf. Subsection 9.6.5).

To construct $\mathcal{C}_\varphi$, we interpose a language projection between two language complementations:

- We complement $\mathcal{B}_\varphi$, that is, we construct an alternating automaton $\mathcal{B}_\varphi^d$ with $\mathcal{L}(\mathcal{B}_\varphi^d) = \overline{\mathcal{L}(\mathcal{B}_\varphi)}$ (Lemma 9.3).
- Next, we build a nondeterministic automaton $\mathcal{N}_\varphi$ with the same language $\mathcal{L}(\mathcal{N}_\varphi) = \mathcal{L}(\mathcal{B}_\varphi^d)$ (Corollary 4.6).
- Then, we construct a nondeterministic automaton $\mathcal{P}_\varphi$ that accepts a $2^{\Pi} \times \mathcal{O}_b \otimes \mathcal{O}_W$-labeled 2^{Π}-tree if it is the $\mathcal{O}_{B \setminus \{b\}}$-projection of a tree accepted by $\mathcal{N}_\varphi$ (Lemma 6.8).

- Finally, we complement $\mathcal{P}_\varphi$, that is, we build an alternating automaton $\mathcal{C}_\varphi$ with $\mathcal{L}(\mathcal{C}_\varphi) = \overline{\mathcal{L}(\mathcal{P}_\varphi)}$ (Lemma 9.3).

An alternating automaton $\mathcal{B} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ can be complemented by dualizing its transition function (that is, replacing each occurrence of $\wedge$, $\vee$, *true* and *false* by $\vee$, $\wedge$, *false* and *true*, respectively) and increasing the color of each state by one (dualizing the acceptance condition). Dualization of alternating automata was introduced by Muller and Schupp [MS87].

Lemma 9.3 *[MS87] Given an alternating automaton $\mathcal{B} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$, the dual automaton $\mathcal{B}^d = (\Sigma, \Upsilon, Q, q_0, \overline{\delta}, \alpha + 1)$, where $\overline{\delta}$ is the function dual to δ, accepts a tree $\langle \Upsilon^*, l \rangle$ if and only if $\langle \Upsilon^*, l \rangle$ is not accepted by $\mathcal{B}$.* □

For a given tree $\langle \Upsilon^*, l \rangle$ in the language of $\mathcal{B}$ and a given winning strategy of player *accept* in the acceptance game for $\langle \Upsilon^*, l \rangle$ (with associated run tree $\langle R, r \rangle$) we can infer a winning strategy for player *reject* for the dual automaton and the same input tree: If, for $\mathcal{B}$, player *accept* chooses a set of atoms upon a given history, than, for the automaton $\mathcal{B}^d$ dual to $\mathcal{B}$, player $reject_{dual}$ commits herself to choose one of these atoms before player $accept_{dual}$ makes his choice (Since the transition functions are dual, the choice of player $accept_{dual}$ cannot be disjoint with the commitment of player $reject_{dual}$). Player $accept_{dual}$ can now enforce only paths of the run tree $\langle R, r \rangle$ of $\langle \Upsilon^*, l \rangle$. Since the strategy of player *accept* is winning (for $\mathcal{B}$), all paths in $\langle R, r \rangle$ satisfy the parity condition for the coloring function α. Consequently, they do *not* satisfy the parity condition for the dual coloring function $\alpha + 1$.

Since dualizing an automaton twice results in the original automaton (with the exception that α is replaced by $\alpha + 2$), each tree accepted by $\mathcal{B}^d$ is rejected by $\mathcal{B}$. Thus, $\mathcal{B}^d$ accepts exactly the trees rejected by $\mathcal{B}$.

Neither the dualization of an automaton (Lemma 9.3), nor language projection (Lemma 6.8) change the set of automata states. The only expensive operation in the construction of $\mathcal{C}$ from $\mathcal{B}$ is the transformation of general alternating tree automata to nondeterministic automata (Corollary 4.6).

Corollary 9.4 *Given an alternating automaton $\mathcal{B} = (\Sigma \times \Xi, \Upsilon, Q, q_0, \delta, \alpha)$ with n states and c colors that runs on $\Sigma \times \Xi$-labeled Υ-trees, we can construct an alternating automaton $\mathcal{C} = (\Sigma, \Upsilon, Q', q_0', \delta', \alpha')$ with $n^{O(c\,n)}$ states and $O(c\,n)$ colors that accepts a Σ-labeled Υ-tree $\langle \Upsilon^*, l_\Sigma \rangle$ if and only if all $\Sigma \times \Xi$-labeled Υ-trees $\langle \Upsilon^*, l_\Sigma \times l_\Xi \rangle$ with $\langle \Upsilon^*, l \rangle = proj_\Sigma(\langle \Upsilon^*, l_\Xi \rangle)$ are accepted by $\mathcal{B}$.* □

9.6.3 Upper Bounds

Our construction establishes 2EXPTIME upper bounds for ATL, CTL, classic μ-calculus and ATμC specifications, a 3EXPTIME upper bound for CTL* specifications, and a 4EXPTIME upper bound for ATL* specifications.

Theorem 9.5 *Checking the resilient realizability* $(\mathbf{A}, \{b\}) \vDash \varphi$ *can, for a given architecture* $\mathbf{A}$*, a given specification* φ *and a given implementation for the white-box processes, be performed in time doubly exponential in the length of* φ *if* φ *is a CTL, ATL, μ-calculus or ATμC specification, and in time triply exponential in the length of* φ *if* φ *is a CTL* specification.*

If applicable, a resilient realization $p_b : (2^{I_b})^* \to \mathcal{O}_b$ *can be constructed within the same complexity bounds.*

Proof: Following the construction described in Section 9.6.1, we build a nondeterministic automaton $\mathcal{G}_\varphi$, which accepts exactly the resilient implementations of φ by b. If $n = |\varphi|$ denotes the length of the specification φ, then $\mathcal{G}_\varphi$ has

- $2^{n^{O(n)}}$ states and $n^{O(n)}$ colors if φ is an ATL (or CTL) specification,
- $2^{n^{O(n^3)}}$ states and $n^{O(n^3)}$ colors if φ is an alternating-time (or classic) μ-calculus specification.
- $2^{2^{2^{O(n)}}}$ states and $2^{2^{O(n)}}$ colors if φ is a CTL* specification, and with

By Theorem 4.7 we can check the emptiness of $\mathcal{G}_\varphi$ and, if $\mathcal{G}_\varphi$ is non-empty, construct a regular tree accepted by $\mathcal{G}_\varphi$ in time polynomial in the number of states and exponential in the number of colors of $\mathcal{G}_\varphi$. □

9.6.4 Lower Bounds

To demonstrate that the upper bounds are sharp, we give a reduction from the synthesis problem in reactive environments with complete information, which is known to be 2EXPTIME and 3EXPTIME hard for CTL and CTL*, respectively [KMTV00]. In synthesis with reactive environments and complete information, we have only one process b, for which a (deterministic) strategy $p_b : (2^{O_{env}})^* \to \mathcal{O}_b$ is sought. The environment can react to the input by restricting its actions to a non-empty subset of its output variables O_{env}, which can be viewed as a non-deterministic strategy $p_{env} : (2^{O_{env} \cup O_b})^* \to 2^{2^{O_{env}}} \setminus \{\emptyset\})$. In our terms, a strategy $p_b : (2^{O_{env}})^* \to \mathcal{O}_b$ implements a specification φ if, for all

strategies $p_{env} : (2^{O_{env} \cup O_b})^* \to 2^{2^{O_{env}}} \setminus \{\emptyset\}$ of the environment, $\langle \| p_b \oplus p_{env} \|, dir \rangle$ is a model of φ.

Synthesis in reactive environments can therefore be viewed as the special case of checking $(\mathbf{A}, \{b\}) \models \varphi$ for an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ with $A = B = \{b, env\}$, $D = \{b\}$, and $I_b = I_{env} = \Pi$.

Note that the determinacy of p_b can, as an alternative to guarantee determinacy by construction, be ensure by strengthening the specification φ such that only deterministic strategies are allowed: We solve the realizability problem for $\varphi' = \varphi \wedge \psi$, with $\psi = \bigwedge_{o \in O_b} \mathrm{AG}\ (\mathrm{EX}o \leftrightarrow \mathrm{AX}o)$, which is linear in φ.

Theorem 9.6 *The realizability problem $(\mathbf{A}, \{b\}) \models \varphi$ is 3EXPTIME-complete for CTL* and 2EXPTIME-complete for CTL, ATL and for classic and alternating-time μ-calculus specifications in the length $|\varphi|$ of the specification.*

Proof: The lower bounds for CTL and CTL* follow from the equal lower bounds for the synthesis problem with reactive environments [KMTV00]. The lower bound for the μ-calculus is established by the lower bound for CTL. The lower bounds for ATμC and ATL follow immediately. The upper bounds are established by Theorem 9.5. □

9.6.5 Universal Specifications

For universal specifications and nondeterministic black-boxes $B \setminus \{b\}$, the quantification step of Subsection 9.6.2 can be simplified, resulting in an exponential improvement in the complexity. A specification is universal if its models are recognized by a universal ACG. The universal specifications include in particular the formulas of the following logics:

- the syntactic subsets ACTL* and ACTL of CTL* and CTL, respectively, which do not contain the existential path quantifier (and allow negation only for atomic propositions),
- the syntactic subset of the modal μ-calculus that does not contain the existential successor operator $\Diamond$, and
- trace languages like LTL.

Intuitively, the quantification step is used to guess, for a given implementation $p_b : (2^\Pi)^* \to \mathcal{O}_b$ of a process b, an implementation $p_{B \setminus \{b\}} : (2^\Pi)^* \to \mathcal{O}_{B \setminus \{b\}}$ for the remaining black-box processes $B \setminus \{b\}$ for which φ_b does *not* hold, that is, $\langle \| p_b \oplus p_W \oplus p_{B \setminus \{b\}} \|, dir \rangle \not\models \varphi_b$. If φ_b is a *universal* specification, the strategy can be set to the constant value $\bigoplus_{b \neq b' \in B} \{2^{O_{b'}}\}$.

Theorem 9.7 *If φ is a universal specification over a set AP of atomic propositions, $\langle Y, l : Y \to 2^{AP}\rangle \not\models \varphi$ is no model of φ, and $\langle Y', l' : Y' \to 2^{AP}\rangle$ is a 2^{AP}-labeled tree with $Y \subseteq Y'$ and $l(y) = l'(y)$ for all $y \in Y$, than $\langle Y', l' : Y' \to 2^{AP}\rangle$ is no model of φ.*

Proof: Let $\mathcal{U}_\varphi$ be a universal ACG recognizing the models of φ, and let r be a winning strategy for player *reject* in the acceptance game of $\langle Y, l : Y \to 2^{AP}\rangle$. (Such a strategy exists, since $\langle Y, l : Y \to 2^{AP}\rangle \not\models \varphi$ is no model of φ.) Then r is also a winning strategy for $\langle Y', l' : Y' \to 2^{AP}\rangle$. ($r$ always picks the same atom and the same direction for $\langle Y', l'\rangle$ as for $\langle Y, l\rangle$, so intuitively $\langle Y, l\rangle$ is never left.) □

Consequently, we can assume without loss of generality that the strategies $p_{b'}$ of all remaining black-box processes $b' \in B \setminus \{b\}$ are the constant strategies that constantly enable all directions.

Corollary 9.8 *If φ is a universal specification and, for all $b \neq b' \in B$, $\mathcal{O}_{b'}$ has a maximal element $\overline{d}_{b'} \in \mathcal{O}_{b'}$ ($\forall d \in \mathcal{O}_{b'}.\, d \subseteq \overline{d}_{b'}$), than $\langle \|p_b \oplus p_W \oplus p_{B\setminus\{b\}}\|, dir\rangle \vDash \varphi$ is a model of φ for all $p_{B\setminus\{b\}} : (2^\Pi)^* \to \mathcal{O}_{B\setminus\{b\}}$ if and only if $\langle \|p_b \oplus p_W \oplus \overline{p}_{B\setminus\{b\}}\|, dir\rangle \vDash \varphi$ is a model of φ for $\overline{p}_{B\setminus\{b\}} : (2^\Pi)^* \to \{\bigoplus_{b\neq b'\in B} \overline{d}_{b'}\}$.* □

For trace languages, it is safe to *extend* $\mathcal{O}_{b'}$ by a maximal element $\overline{d}_{b'} = \bigcup_{d\in\mathcal{O}_{b'}} d$ for every $b' \in B \setminus b$, since no new traces are introduced by this extension. That is, for $\overline{p}_{B\setminus\{b\}} : (2^\Pi)^* \to \{\bigoplus_{b'\in B\setminus\{b\}} \overline{d}_{b'}\}$, each trace in $\|p_b \oplus p_W \oplus \overline{p}_{B\setminus\{b\}}\|$ is a trace in $\|p_b \oplus p_W \oplus p_{B\setminus\{b\}}\|$ for some $\{p_{b'} : (2^\Pi)^* \to \mathcal{O}_{b'} \mid b' \in B \setminus \{b\}\}$.

This is, however, *not* true for general universal specifications: If the set of black-box processes $B = \{b, b'\}$ consists of two processes b and b' and we allow only for deterministic implementations of b', then b can resiliently realize $(\mathrm{AX}p) \vee (\mathrm{AX}\neg p)$. Obviously, this is no longer true if we add a maximal element to $\mathcal{O}_{b'}$, which simultaneously allows for successors where p holds and for successors where p does not hold.

9.6.6 Premise (S)

The correctness of Premise (S) can be checked along the same lines: We check whether the empty strategy of a shadow process (without in- or output variables) resiliently realizes $\bigwedge_{b\in B} \varphi_b \to \psi$. Since $\mathcal{O}_\emptyset = \{\emptyset\}$ and $I_\emptyset = \emptyset$, the automaton $\mathcal{F}$ (with n states and c colors) is an alternating word automaton over the single-letter alphabet, whose emptiness can be checked in $n^{O(c)}$ time.

Theorem 9.9 *Checking Premise (S) is 2EXPTIME-complete for CTL* specifications, and EXPTIME-complete for CTL, ATL, and classic and alternating-time μ-calculus specifications $\varphi = \bigwedge_{b \in B} \varphi_b \rightarrow \psi$ in the length of φ.*

Proof: To be able to argue independent from the structure of the transition function of $\mathcal{F}$, we can check Premise (S) by an adjusted algorithm that postpones the quantification over the relaxed strategies of the black-box processes till after pruning the directions and white-box implementations from the label. Then $\mathcal{F}$ is empty if and only if $\mathcal{E}$ is empty, and $\mathcal{E}$ is now dual to a nondeterministic automaton $\mathcal{P}$ that is exponential in the size of $\mathcal{A}$.

With Theorem 4.7 and the complexity of translating temporal specifications to ATμC [AHK02, BC96], we get EXPTIME upper bounds for CTL, ATL, and classic and alternating-time μ-calculus specifications, and a 2EXPTIME bound for CTL* specifications, respectively.

Fitting lower bounds can be obtained by reduction from module checking [KV97a]. □

9.7 Conclusions

We have introduced a sound and complete proof rule for distributed synthesis, which reduces the distributed synthesis problem to the simpler task of deciding the *resilient realizability* of *local* process specifications.

Synthesizing resilient implementations generalizes the synthesis of open systems. Open synthesis assumes an environment with *maximal* behavior. For resilient implementations, this environment model is extended in two aspects: (1) The other black-box processes add a *reactive* component to the environment, and (2) the process only has *incomplete information* about the environment behavior.

Extension (1) is expensive. Adding the reactive component increases the complexity for CTL specifications from EXPTIME [KV99] to 2EXPTIME [KMTV00], and for CTL* specifications from 2EXPTIME [KV99] to 3EXPTIME [KMTV00]. As shown in Section 9.6, extension (2) has no extra cost. This settles an open question of [KMTV00]: The complexity of synthesizing a single process in a distributed architecture is still 2EXPTIME-complete and 3EXPTIME-complete, respectively.

The semi-automatic compositional approach is an efficient alternative to fully automatic distributed synthesis. Distributed synthesis is only decidable for

a restricted class of architectures. For this class of architectures, the complexity is nonelementary [PR90, KV01] (cf. Chapter 6).

The situation is similar to the *verification* of distributed systems, where the compositional approach is well-established [dRLP98]. Our proof rule is a first example of a compositional synthesis technique. The rule is complete and therefore sufficient to decompose any realizable specification. The rule may, however, be less convenient to use than some compositional verification rules that, for example, apply circular assume-guarantee reasoning [Mai03]. Defining such rules for the synthesis problem is an interesting topic of future research.

Chapter 10

Asynchronous Systems

10.1 Introduction

Synthesis automatically transforms a specification into an implementation that is guaranteed to satisfy the specification. For *synchronous* systems, the synthesis problem is well-understood. Synthesizing single-process implementations is EXPTIME-complete for the μ-calculus [Koz83, KV00], and 2EXPTIME-complete for linear-time temporal logic (LTL) and computation-tree logic (CTL*) [KV97b, KV00, KMTV00]. Multi-process synthesis, the problem of finding implementations for the processes in a given distributed architecture, has been solved for pipelines [PR90], rings [KV01], and in general for all architectures without information forks (cf. Chapter 6).

By contrast, the problem of synthesizing *asynchronous* systems has so far received very little attention: The synthesis algorithms in the literature are limited to LTL specifications and single-process implementations. The first solution for asynchronous synthesis with specifications in LTL, but without fairness conditions, is due to Pnueli and Rosner [PR89b]. Anuchitanukul and Manna [AM94] later showed that fairness conditions can be included in a deductive approach; Vardi [Var95] provided an automata-based algorithm for the same problem.

The question arises if the lack of synthesis algorithms for asynchronous systems is a coincidence or rather an indication of an inherent hardness of the synthesis problem for asynchronous systems. In this chapter, we systematically study the challenges in extending synthesis to the asynchronous case and, in doing so, give a comprehensive answer to this question.

Challenge 1: Synthesizing asynchronous processes for branching-time specifications. We begin by generalizing the synthesis of single-process implementations from linear-time to branching-time[1] specifications. The behavior of an asynchronous process depends on the scheduler: While synchronous processes are aware of each change to their inputs, asynchronous processes may fail to see certain changes (when the writing process is scheduled more often than the reading process) and may see duplicate input values (when the reading process is scheduled multiple times between two writes). For linear-time specifications, asynchronous processes are typically analyzed in combination with a *full* scheduler, which allows every possible scheduling to occur along some path of the computation tree. In our first algorithm, we adapt this setting to branching-time specifications and synthesize an asynchronous process implementation such that the computation tree that results from the combination with a full scheduler satisfies the branching-time specification. The algorithm runs in exponential time for μ-calculus specifications and in double exponential time for CTL*. We thus obtain the result that *under full scheduling, the cost of synthesizing single-process implementations is the same for synchronous and asynchronous systems.*

Challenge 2: Synthesizing scheduler-independent implementations. Dropping the assumption of a full scheduler leads to the problem of synthesizing *scheduler-independent* implementations: We require that the implementation must satisfy the specification for *every* scheduler. For LTL (and, more generally, for universal specifications), the two synthesis problems coincide. For branching-time specifications, scheduler-independent synthesis is the strictly more general problem. Consider the existential specification "there is a path where the output of the process changes in every second step." This specification can trivially be satisfied under the assumption of a full scheduler, but there is no implementation that guarantees this specification for all schedulers. Scheduler-independent synthesis allows us to explicitly state the assumptions on the scheduler as part of the specification. An interesting example for such an assumption is fairness. While synthesis under full scheduling allows us to find implementations that perform correctly on *fair paths* ("there is a *fair scheduling* where the output of the process changes in every second step"), scheduler-independent synthesis allows us to find implementations that perform correctly whenever the scheduler

[1] The presented techniques extend to alternating-time specification language. The restriction to branching-time logic is chosen to avoid a discussion on the appropriated semantics, in particular the question if the scheduler is to be treated like an ordinary process, and if it can form coalitions.

is fair ("if the *scheduler* is *fair* on all paths then there is a path where the output of the process changes in every second step"). In our second algorithm, we synthesize an asynchronous process implementation such that *any* computation tree that results from the combination of the process with some scheduler satisfies the branching-time specification. The algorithm runs in double exponential time for CTL and μ-calculus specifications, and in triple exponential time for CTL*. We provide matching lower bounds for these logics, obtaining the result that *scheduler-independent synthesis is exponentially harder than synthesis under full scheduling.*

Challenge 3: Synthesizing asynchronous distributed systems. We consider the multi-process synthesis problem, where the distributed architecture is, as usual, given as a directed graph. In the synchronous case, the distributed synthesis problem is decidable if and only if the architecture does not contain an information fork (cf. Chapter 6). We show that in the asynchronous case, *the distributed synthesis problem is decidable if and only if the architecture contains only a single black-box process.*

Our results thus demonstrate that, except for the case of single-process implementations and full scheduling, the synthesis of asynchronous systems is indeed harder than the synthesis of synchronous systems. Our algorithms solve the distributed synthesis problem for architectures with a single black-box process. Since the synthesis problem is undecidable for all architectures with two or more black-box processes, it is impossible to extend our algorithms to a larger set of architectures.

Challenge 4: Synthesizing globally asynchronous but locally synchronous systems. We finally consider systems that are composed globally asynchronously, but that have local islands of synchronized processes (GALS systems [Gup03]). We combine the decision procedure for systems of asynchronously composed components with a single black-box with the decision procedure for synchronous systems of Chapter 6 to a decision procedure for GALS systems that covers all GALS systems that satisfy two constraints:

- all black-box components belong to the same synchronized component (that is, they are always scheduled together), and
- this synchronized area does not contain an information fork.

It is then simple to combine the respective completeness results to show that any GALS system that violates either requirement immediately becomes undecidable.

10.2 The Synthesis Problem

We study the synthesis problem in the general setting of distributed systems. The *synthesis problem* is to decide for the triple $(\mathbf{A}, \varphi, \{p_w\}_{w \in W})$, consisting of an architecture $\mathbf{A}$, a specification φ, and a set of white-box strategies $\{p_w\}_{w \in W}$, whether there exists an implementation $\{p_b\}_{b \in B}$ for each black-box process in $\mathbf{A}$, such that their computation tree $\langle \| p_A \oplus sched \|, dir \rangle$ satisfies φ. The *scheduler* *sched* is a special process that selects, in each turn, the subset of scheduled processes of $\mathbf{A}$. Processes that are not scheduled are not aware of this fact. That is, the scheduler adds an additional source of incomplete information to the distributed synthesis problem.

10.2.1 The Scheduler

In every step, the scheduler makes a (possibly nondeterministic) choice which processes are scheduled. In a *full* scheduler, all choices are possible in each step. In general, some choices may be disabled, and the set of choices may depend on the history of states.

We formalize the scheduler as a function from $(2^{A \cup \Pi})^*$ to the set of potential scheduling decisions $\mathcal{O}_{sched} = 2^{2^A}$, which consists of the set of subsets of the set of processes. We represent the function as an $\mathcal{O}_{sched}$-labeled $2^{A \cup \Pi}$-tree $\langle (2^{A \cup \Pi})^*, sched \rangle$, where the label refers to the nondeterministic choice of the scheduler.

The scheduler can be viewed as a fully informed process that has access to the complete history of decisions. It is better informed than all other processes, because even if a process $a \in A$ has access to all system variables, a cannot observe their values unless it is scheduled.

10.2.2 Computations

The computation tree identifies the system state (that is, the values of the system variables and the currently scheduled processes) for every possible history of input assignments and scheduling decisions. As an intermediate construction, we define, for each process $a \in A$, the function $vis_a : (2^{A \cup \Pi})^* \to (2^{I_a})^*$ that

maps a global history of system states to the input history available to a. We set

- $vis_a(\varepsilon) = \varepsilon$,
- $vis_a(x \cdot (A', \pi)) = vis_a(x) \cdot (\pi \cap I_a)$ if $a \in A'$ and
- $vis_a(x \cdot (A', \pi)) = vis_a(x)$ if $a \notin A'$.

For asynchronous systems, we define the *computation tree* for an implementation $P = \{p_a\}_{a \in A}$ and a scheduler *sched* as the greatest total $2^{A \cup \Pi}$-labeled $2^{A \cup \Pi}$-tree $\langle Y_{ct}, dir \rangle$ such that $y \cdot (A', \pi) \in Y_{ct}$ implies

- $A' \in c(y)$ – the scheduling decision is possible,
- $\forall a \notin A'. O_a \cap \pi = O_a \cap dir(y)$ – the output variables of non-scheduled processes remains unchanged, and
- $\forall a \in A'. O_a \cap \pi \in p_a(vis_a(y))$ – the output of the scheduled processes is in accordance with their implementation.

As in the synchronous setting, it is technically convenient to use a relaxed implementation $p_a : (2^{A \cup \Pi})^* \to \mathcal{O}_a$ that has access to the system state in intermediate steps of the synthesis algorithm.

10.2.3 The Synthesis Problem

A triple $(\mathbf{A}, \varphi, \{p_w\}_{w \in W})$, consisting of an architecture $\mathbf{A}$, a specification φ, and a set of white-box implementations $P_W = \{p_w\}_{w \in W}$, is called

- *realizable under full scheduling* if there exists an implementation $P_B = \{p_b\}_{b \in B}$ of the black-box processes such that, for the full scheduler that maps every input history to 2^A, the resulting computation tree $\langle Y_{ct}, dir \rangle$ is a model of φ; and
- *scheduler-independently realizable* if there exists an implementation $P_B = \{p_b\}_{b \in B}$ of the black-box processes such that, for all schedulers $\langle (2^{A \cup \Pi})^*, sched \rangle$, the resulting computation tree $\langle Y_{ct}, dir \rangle$ is a model of φ.

We call an architecture $\mathbf{A}$ (scheduler-independently) *decidable* if an algorithm exists that decides for all specifications φ and all sets of finite-state white-box implementation $\{p_w\}_{w \in W}$ if $(\mathbf{A}, \varphi, \{p_w\}_{w \in W})$ is (scheduler-independently) realizable.

10.3 Single-Process Synthesis under Full Scheduling

In this section, we show that, under the assumption of full scheduling, the cost of synthesizing single-process implementations is the same for synchronous and asynchronous systems. We develop an automata-theoretic synthesis algorithm for asynchronous systems with a single black-box process. The algorithm runs in time exponential in the length of a CTL or μ-calculus specification, and in time doubly exponential in the length of a CTL* specification, respectively.

10.3.1 Overview

The algorithm assumes an architecture $\mathbf{A}$ with a single black-box process b. It starts by representing a specification φ as a symmetric alternating automaton $\mathcal{A}_\varphi$, which is transformed into a nondeterministic automaton $\mathcal{F}_\varphi$ that accepts a tree $\langle (2^{I_b})^*, p_b \rangle$ if and only if p_b is an implementation of φ. The solution of the emptiness game for $\mathcal{F}_\varphi$ then provides such an implementation.

The algorithm consists of the following automata constructions:

- **From formulas to automata.** First, we construct the symmetric alternating automaton[2] $\mathcal{A}_\varphi$ that accepts exactly the models of φ (Theorem 4.1 and Lemma 4.2).

- **From models to relaxed implementations.** In a second step, we build an alternating automaton $\mathcal{B}_\varphi$ that accepts a relaxed implementation (including the scheduler) $\langle (2^{A \cup \Pi})^*, l \times sched \oplus \bigoplus_{a \in A} p_a^r \rangle$ if the tree $\langle Y_{ct}, l \rangle$ defined by a labeling function $l : (2^{A \cup \Pi})^* \to 2^{A \cup \Pi}$, the scheduler *sched* and the relaxed implementation $P = \{p_a^r\}_{a \in A}$ is accepted by $\mathcal{A}_\varphi$ (Lemma 6.4).

- **Pruning the scheduler.** We then construct an alternating automaton $\mathcal{C}_\varphi$ that accepts a relaxed implementation $\langle (2^{A \cup \Pi})^*, l \times \bigoplus_{a \in A} p_a^r \rangle$ if its extension $\langle (2^{A \cup \Pi})^*, l \times sched \oplus \bigoplus_{a \in A} p_a^r \rangle$ by the decisions of the full scheduler[3] $sched : (2^{A \cup \Pi})^* \to \mathcal{O}_{sched}$ is accepted by $\mathcal{B}_\varphi$ (Lemma 6.5).

- **Adjusting for white-box processes.** We then build an alternating automaton $\mathcal{D}_\varphi$ that accepts a relaxed implementation $\langle (2^{A \cup \Pi})^*, l \times p_b^r \rangle$ of b if its extension $\langle (2^{A \cup \Pi})^*, l \times \bigoplus_{a \in A} p_a^r \rangle$ with the relaxed implementations of

[2] Symmetric alternating automata [Wil01] are a simpler variant of automata over concurrent game structures that only branch to all or to some successor.

[3] We can use any *known* scheduler instead of the full scheduler.

the white-box processes is accepted by $\mathcal{C}_\varphi$. This is done by first translating implementations to relaxed implementations (Lemma 10.1), and then pruning the white-box decisions from the label (Lemma 6.5).

- **From relaxed implementations to implementations.** We then construct the alternating automaton $\mathcal{E}_\varphi$ that accepts an implementation $\langle (2^{I_b})^*, p_b \rangle$ if and only if $\langle (2^{A \cup \Pi})^*, dir \times p_b^r \rangle$ is, for the related relaxed implementation p_a^r of $p_b = p_b^r \circ vis_b$, accepted by $\mathcal{D}_\varphi$ (Lemmata 10.2 and 10.3).
- **Strategy construction.** Finally, we build a nondeterministic automaton $\mathcal{F}_\varphi$, with $\mathcal{L}(\mathcal{F}_\varphi) = \mathcal{L}(\mathcal{E}_\varphi)$ (Corollary 4.6), and construct a strategy for the black-box process such that the induced computation tree is a model of φ (or demonstrate that no such strategy exists) by solving the emptiness game for $\mathcal{F}_\varphi$ (Theorem 4.7).

10.3.2 Adjusting for white-box processes

Eliminating the white-box decisions from the label consists of two steps: First, the implementations are translated into relaxed implementations (Lemma 10.1), and then the white-box implementations are eliminated using the construction from Chapter 6 (Lemma 6.5).

Translating known implementations to relaxed implementations is simple. We assume that the implementations are given as a family of deterministic finite-state Moore machines $\{\mathcal{M}_w = (2^{I_w}, S_w, s_0^w, d_w, o_w)\}_{w \in W}$ with

- input alphabet 2^{I_w},
- a finite set S_w of states with initial state s_0^w,
- transition function $d_w : 2^{I_w} \times S \to S$, and
- output function $o_w : S_w \to \mathcal{O}_w$ that maps each state $s \in S_w$ of $\mathcal{M}_w$ to the (possibly nondeterministic) choice of the process w in s

that capture the behavior of the respective white-box process w.

Lemma 10.1 *Given a deterministic finite-state Moore machine* $\{\mathcal{M}_w = (2^{I_w}, S_w, s_0^w, d_w, o_w)\}_{w \in W}$ *that captures the behavior of* w *on the visible input, we can construct a deterministic finite-state Moore machine* $\{\mathcal{M}'_w = (2^{A \cup \Pi}, S_w, s_0^w, d'_w, o_w)\}_{w \in W}$ *that captures the behavior of* w *on the relaxed input.*

Proof: It suffices to ignore inputs that are not visible to w, and to use only the I_b part of inputs from rounds where w is scheduled; that is, to set

- $d'_w : \big((A', \pi), s\big) \mapsto d_w(\pi \cap I_b, s)$ if $w \in A'$, and
- $d'_w : \big((A', \pi), s\big) \mapsto s$ otherwise.

By construction, $\mathcal{M}'_w$ reaches a state s upon the input sequence $I_1 I_2 I_3 \ldots \in (2^{A \cup \Pi})^*$ if and only if $\mathcal{M}_w$ reaches s upon the input sequence $vis_w(I_1 I_2 I_3 \ldots) \in (2^{I_w})^*$. □

10.3.3 From Relaxed Implementations to Implementations

The central automata transformation for asynchronous synthesis is the transformation of an alternating automaton $\mathcal{D}_\varphi$ that accepts a relaxed implementation p_b^r for a black-box process b into an automaton $\mathcal{E}_\varphi$ that accepts an implementation p_b if and only if $p_b \circ vis_b$ is accepted by $\mathcal{D}$.

Comparable to Lemma 6.7, this transformation reduces the information available to b. However, the reduction is more involved, because we face two sources of incompleteness at the same time: The incompleteness through the limited input alphabet (b sees only the variables in I_b), and the incompleteness caused by the scheduler.

As an intermediate step, we construct an alternating ε-automaton $\mathcal{E}_\varphi^\varepsilon$, which extends the alternating automata introduced in Subsection 4.2.3 by ε-transitions (cf. Section 4.3). For technical convenience, the pruning of directions from the label is moved to this construction.

Lemma 10.2 *For a given alternating automaton $\mathcal{D} = (2^{A \cup \Pi} \times \mathcal{O}_b, 2^{A \cup \Pi}, Q, q_0, \delta, \alpha)$, we can construct an alternating ε-automaton $\mathcal{E}^\varepsilon = (\mathcal{O}_b, 2^{A \cup \Pi}, Q \times 2^{A \cup \Pi}, (q_0, \upsilon_0), \gamma, \alpha')$ that accepts a tree $\langle (2^{I_b})^*, p_b \rangle$ if and only if $\langle (2^{A \cup \Pi})^*, dir \times p_b \circ vis_b \rangle$ is accepted by $\mathcal{D}$.*

Proof: In a first step, we move the directions from the label to the states (Lemma 6.6), resulting in the alternating automaton $\mathcal{D}' = (\mathcal{O}_b, 2^{A \cup \Pi}, Q \times 2^{A \cup \Pi}, (q_0, \upsilon_0), \delta', \alpha')$. Note that the $2^{A \cup \Pi}$ component of the state now always reflects the direction of the current node of the input tree.

We then use the function $\beta : 2^{A \cup \Pi} \to I_b \cup \{\varepsilon\}$ with

- $\beta : (A', \pi) \to \pi \cap I_b$ if $b \in A'$, and
- $\beta : (A', \pi) \to \{\varepsilon\}$ if $b \notin A'$

as the central auxiliary function for the construction. β can be used to stepwise transform the input to the relaxed implementation to the input visible to b: For $I_1 I_2 I_3 \ldots \in (2^{A \cup \Pi})^*$, $vis_b(I_1 I_2 I_3 \ldots) = \beta(I_1)\beta(I_2)\beta(I_3) \ldots \in (2^{I_b})^*$ holds true.

Let $b^{\mathbf{O}_b}_{(q;A',\pi)}\big(\{(q_i; A'_i, \pi_i), (A'_i, \pi_i)\}_{i \in I}\big) = \delta'\big((q; A', \pi), \mathbf{O}_b\big)$ be the positive Boolean function defined by $\delta'\big((q; A', \pi), \mathbf{O}_b\big)$. Using β, we can construct $\mathcal{E}^\varepsilon$ such that $\mathcal{E}^\varepsilon$ simulates the behavior of $\mathcal{D}'$ on the relaxed implementation $\langle (2^{A \times \Pi})^*, p_b \circ vis_b \rangle$ when $\mathcal{E}^\varepsilon_\varphi$ reads an input tree $\langle (2^{I_b})^*, p_b \rangle$ by setting $\gamma\big((q; A', \pi), \mathbf{O}_a\big) = b^{\mathbf{O}_b}_{(q;A',\pi)}\big(\{(q_i; A'_i, \pi_i), \beta(A'_i, \pi_i)\}_{i \in I}\big)$. □

$\mathcal{E}^\varepsilon_\varphi$ can be transformed into an ordinary (ε-free) alternating automaton $\mathcal{E}_\varphi$ by a construction similar to the ε-elimination for automata over concurrent game structures (Lemma 4.2).

Lemma 10.3 *[Wil99, KV00] Given an alternating ε-automaton $\mathcal{E}^\varepsilon$ with n states and c colors, we can construct an ε-free alternating automaton $\mathcal{E}$ with at most $O(c\,n)$ states and c colors.* □

10.3.4 Complexity

The construction described in Subsection 10.3.1 provides EXPTIME and 2EXPTIME upper bounds for the synthesis problem under full scheduling in case of μ-calculus and CTL* specifications, respectively. Matching lower bounds can be inferred from the known lower bounds for the synthesis problems for CTL and CTL* in synchronous systems by applying linear specification transformations.

Theorem 10.4 *The distributed synthesis problem under full scheduling for architectures with a single black-box process is EXPTIME-complete for specifications in CTL and the μ-calculus, and 2EXPTIME-complete for CTL* specifications.*

Proof: The upper bounds follow from the construction suggested in Subsection 10.3.1.

For the lower bounds, we consider again the simpler setting of environment synthesis, that is, we only allow for architectures $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ with $A = \{env, b\}$, $B = D = \{b\}$ and $I_b = O_{env}$, and fix the implementation of the environment process *env* to be unconstrained; that is, p_{env} maps every input history to $2^{O_{env}}$.

We reduce the synthesis problem for this case from the synthesis problem for the synchronous setting, which is EXPTIME-hard for CTL and 2EXPTIME-hard for CTL* [KV99]. The reduction is by a linear transformation of each CTL

or CTL* formula φ_{sync} that reasons only over the communication variables Π, to a CTL or CTL* specification φ_{async}, respectively, such that φ_{async} is realizable if and only if φ_{sync} if realizable in the synchronous setting.

For CTL specifications, we replace every occurrence of $A\varphi U\psi$, $E\varphi U\psi$, $AX\psi$, and $EX\psi$ by $A\varphi U(\psi \vee \neg b \vee \neg env)$, $E(\varphi \wedge b \wedge env)U(\psi \wedge b \wedge env)$, $AX(b \wedge env \rightarrow \psi)$, and $EX(b \wedge env \wedge \psi)$, respectively.

For CTL* specifications, we replace every occurrence of $A\pi$ by $A(G(b \wedge env) \rightarrow \pi)$ and every occurrence of $E\pi$ by $E(G(b \wedge env) \wedge \pi)$.

A strategy p_b for the black-box process is obviously a realization for the transformed specification if and only if p_b realizes the original specification in the synchronous setting.

Finally, the EXPTIME hardness for CTL implies EXPTIME hardness for the μ-calculus, and the EXPTIME upper bound for the μ-calculus establishes a matching upper bound for CTL. □

10.4 Synthesis of Scheduler-Independent Implementations

We now present an algorithm for *scheduler-independent* synthesis, where we only consider implementations that satisfy the specification for *all* schedulers. Scheduler-independent synthesis can also be used to find implementations that satisfy their specification if the scheduler satisfies assumptions that are explicitly stated in the specification.

10.4.1 Overview

We again begin with an overview over the main steps of the construction. The algorithm runs in 2EXPTIME and 3EXPTIME in the length of a μ-calculus and CTL* specification, respectively.

- **From formulas to automata.** First, we construct the symmetric alternating automaton $\mathcal{A}_\varphi$ that accepts exactly the models of φ (Theorem 4.1 and Lemma 4.2).

- **From models to relaxed implementations.** In a second step, we construct an alternating automaton $\mathcal{B}_\varphi$ that accepts a relaxed implementation (including the scheduler) $\langle (2^{A \cup \Pi})^*, l \times sched \oplus \bigoplus_{a \in A} p_a^r \rangle$ if the tree $\langle Y_{ct}, l \rangle$ defined by a labeling function $l : (2^{A \cup \Pi})^* \rightarrow 2^{A \cup \Pi}$, the scheduler

sched and the relaxed implementation $P = \{p_a^r\}_{a \in A}$ is accepted by $\mathcal{A}_\varphi$ (Lemma 6.4).

- **Considering all schedulers.** We then build an alternating automaton $\mathcal{C}_\varphi$ that accepts a relaxed implementation $\langle (2^{A \cup \Pi})^*, l \times \bigoplus_{a \in A} p_a^r \rangle$ if, for *all* schedulers $sched : (2^{A \cup \Pi})^* \to \mathcal{O}_{sched}$, its extension $\langle (2^{A \cup \Pi})^*, l \times sched \oplus \bigoplus_{a \in A} p_a^r \rangle$ by the decisions of the scheduler is accepted by $\mathcal{B}_\varphi$ (Corollary 9.4).

- **Adjusting for white-box processes.** We then construct an alternating automaton $\mathcal{D}_\varphi$ that accepts a relaxed implementation $\langle (2^{A \cup \Pi})^*, l \times p_b^r \rangle$ if its extension $\langle (2^{A \cup \Pi})^*, l \times \bigoplus_{a \in A} p_a^r \rangle$ with the relaxed implementations of the white-box processes is accepted by $\mathcal{B}_\varphi$ (Lemmata 10.1 and 6.5).

- **From relaxed implementations to implementations.** We then build the alternating automaton $\mathcal{E}_\varphi$ that accepts an implementation $\langle (2^{I_b})^*, p_b \rangle$ if and only if $\langle (2^{A \cup \Pi})^*, dir \times p_b^r \rangle$ is, for the related relaxed implementation p_a^r of $p_b = p_b^r \circ vis_b$, accepted by $\mathcal{D}_\varphi$ (Lemmata 10.2 and 10.3).

- **Strategy construction.** Finally, we construct a nondeterministic automaton $\mathcal{F}_\varphi$, with $\mathcal{L}(\mathcal{F}_\varphi) = \mathcal{L}(\mathcal{E}_\varphi)$ (Corollary 4.6), and construct a strategy for the black-box process such that the induced computation tree is a model of φ (or demonstrate that no such strategy exists) by solving the emptiness game for $\mathcal{F}_\varphi$ (Theorem 4.7).

10.4.2 Complexity

The construction provides 2EXPTIME, 3EXPTIME, and 2EXPTIME upper bounds in the length of a specification for the scheduler-independent synthesis problem for CTL, CTL*, and μ-calculus specifications, respectively.

The 2EXPTIME and 3EXPTIME hardness, respectively, of scheduler-independent realizability checking for CTL and CTL* specifications, respectively, can be obtained by a reduction from the CTL and CTL* synthesis problem, respectively, for synchronous systems in reactive environments [KMTV00].

Theorem 10.5 *The scheduler-independent realizability and synthesis problem is 2EXPTIME-complete for CTL and μ-calculus specifications, and 3EXPTIME-complete for specifications in CTL*.*

Proof: The upper bounds follow from the construction suggested in this section. For the lower bounds, we consider again the simpler setting of environment synthesis, that is, we again allow only for architectures $\mathbf{A} =$

$(A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ with $A = \{env, b\}$, $B = D = \{b\}$ and $I_b = O_{env}$, and fix the implementation of the environment process *env* to be unconstrained; that is, p_{env} maps every input history to $2^{O_{env}}$.

To establish the lower bound for CTL* specifications, we transform a CTL* specification φ, which reasons only over the communication variables Π, into a CTL* specification ψ_φ, which is scheduler-independent realizable if and only if φ is realizable in a synchronous setting with a reactive environment. An environment is called reactive [KMTV00] if it can disable a subset (but not all) of its responses in each turn. A full $2^{O_b \cup I_b}$-labeled 2^{I_b}-tree $\langle (2^{I_b})^*, l\rangle$ is a realization of φ in a reactive environment if and only if every total subtree of $\langle (2^{I_b})^*, l\rangle$ is a model of φ and the 2^{I_b}-projection of $\langle (2^{I_b})^*, l\rangle$ is a tree, where every node is labeled with its direction ($proj_{2^{I_b}}(\langle (2^{I_b})^*, l\rangle) = \langle (2^{I_b})^*, dir\rangle$).

Our transformation puts three assumptions on the scheduler: First, we assume that the environment is always scheduled ($\alpha_1 = AG\, p_{env}$). Then, we assume that the process b is scheduled initially and, once it is not scheduled, is never scheduled again ($\alpha_2 = b \wedge A\, b\, U\, AG \neg b$). And finally, we assume that if b is scheduled, then there is a path where b as always scheduled ($\alpha_3 = AG\,(b \rightarrow EG\, b)$).

There is a natural bijection between total 2^{I_b}-trees and schedulers that satisfy these assumptions: We simply map a total 2^{I_b}-tree Y to the scheduler $\langle (2^{A\cup\Pi})^*, sched_Y\rangle$ that always schedules the environment, and that schedules b if and only if the 2^{I_b} projection of the input sequence is in Y:

$$b \in sched_Y(y) \Leftrightarrow hide_{2^{(A\cup\Pi)\setminus I_b}}(y) \in Y.$$

This choice of the scheduler results in a computation tree Y_{ct} that is isomorphic to Y. Consequently, if we transform a CTL* specification φ to a specification φ' by replacing all quantifications over all paths/some path by quantifications over all paths/some path, where b is constantly scheduled, an implementation $\langle (2^{I_b})^*, p_b\rangle$ realizes φ for a given total tree Y in the synchronous setting if and only if it realizes φ' for the scheduler $sched_Y$. $\langle (2^{I_b})^*, p_b\rangle$ is therefore a realization of φ in a synchronous setting with a reactive environment if and only if it is a realization of φ' for all schedulers that satisfy the assumptions α_1, α_2 and α_3. This is equivalent to realizing $\psi_\varphi = (\alpha_1 \wedge \alpha_2 \wedge \alpha_3) \rightarrow \varphi'$ for all schedulers.

Since φ' can be obtained from φ by replacing each occurrence of $A\pi$ and $E\pi$ in φ by $A(Gb \rightarrow \pi)$ and $E(Gb \wedge \pi)$, respectively, the length of ψ_φ is linear in the length of φ. The 3EXPTIME hardness of scheduler-independent realizability checking for CTL* specifications therefore follows from the 3EXPTIME hard-

ness of realizability checking for CTL* specifications in a synchronous setting with a reactive environment [KMTV00].

To also obtain 2EXPTIME-hardness for CTL specifications, we can translate φ to φ' by replacing

- each occurrence of $AX\psi$ by $AX\,(b \rightarrow \psi)$,
- each occurrence of $EX\psi$ by $EX\,(b \wedge \psi)$,
- each occurrence of $A\,\psi_1\,U\,\psi_2$ by $A\,\psi_1\,U\,(b \rightarrow \psi_2)$, and
- each occurrence of $E\,\psi_1\,U\,\psi_2$ by $E\,(b \wedge \psi_1)\,U\,(b \wedge \psi_2)$.

Finally, the 2EXPTIME hardness of realizability checking for CTL specifications implies the 2EXPTIME hardness of realizability checking for μ-calculus specifications. □

10.4.3 Synthesis with Explicit Assumptions on the Scheduler

We close the discussion of scheduler-independent synthesis with the remark that this type of synthesis can also be used to find implementations that satisfy a specification φ as long as the scheduler satisfies an explicitly stated *assumption* α: We simply weaken the specification φ to $\varphi' = \alpha \rightarrow \varphi$.

The assumption α might, for example, specifically specify a round-robin scheduler. The most common assumption on schedulers, however, is *fairness*: A scheduling is considered *impartial* towards a process p if p is scheduled infinitely often, *just* if p is infinitely often disabled or scheduled, and *compassionate* if p being enabled infinitely often implies that p is scheduled infinitely often. The enabledness $\mathit{enabled}(p)$ of a process $p \in A$ can be expressed using new output variables for the respective processes (without changing the input). Quantifying over all fair schedulers for a specification φ is equivalent to quantifying over all schedulers for a modified specification φ' that is satisfied both if φ is satisfied or if the scheduler is not fair. With the fairness condition expressed as a path formula (for example, justice is expressed by $\pi_p = GF\neg\mathit{enabled}(p) \vee GFp$), we obtain the fairness constraint $\pi = \bigwedge_{p \in A} \pi_p$. The modified specification φ' is the implication $\varphi' = (A\pi) \rightarrow \varphi$.

Synthesis under full scheduling and scheduler-independent synthesis thus give us two different approaches to deal with fairness assumptions. While synthesis under full scheduling allows us to require that a property shall hold for

all fair *schedules* (by replacing all occurrences of $A\psi$ and $E\psi$ in CTL* specifications by $A(\pi \rightarrow \psi)$ and $E(\pi \wedge \psi)$, respectively), scheduler-independent synthesis allows us to require that a property hold for all fair *schedulers*.

10.5 Multi-Process Synthesis

The algorithms from Sections 10.3 and 10.4 solve the synthesis problem for all architectures with a single black-box process. We now show that the synthesis problem is undecidable for all architectures with more than one black-box process. Our synthesis algorithms thus cover all decidable asynchronous architectures.

The following theorem states the undecidability result for synthesis under full scheduling; the undecidability of scheduler-independent synthesis follows as a corollary.

Theorem 10.6 *The synthesis problem is undecidable for all architectures with at least two black-box processes and CTL or LTL as specification language.*

Proof: We prove undecidability by a reduction from Post's Correspondence Problem (PCP). For a given alphabet A, an instance of PCP consists of an indexed set of pairs of words $(u_i, v_i), u_i, v_i \in A^+, i \in I = \{1, \ldots, n\}$, over an alphabet A. A solution of PCP is a sequence of indices $i_1, i_2, \ldots, i_m \in I^+$ such that $u_{i_1} \cdot u_{i_2} \cdot \ldots \cdot u_{i_m} = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_m}$.

For simplification, we consider architectures $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a \in A}, \{O_a\}_{a \in A})$ with $A = B = D = \{p, q\}$ that contain only two deterministic black-box processes p and q.

The basic idea of the reduction is to let process p compute the sequence of indices $i_1, i_2, \ldots, i_m$, and to let q produce the corresponding word $u_{i_1} \cdot u_{i_2} \cdot \ldots \cdot u_{i_m} = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_m}$. To check that the word produced by q corresponds to the sequence produced by p, we consider two different schedulings, one in which p produces the indices along the u-words, and one in which p produces the indices along the v-words.

We ensure that the two processes always see the constant input 0 along both paths, and must therefore produce the same output on them.

Each index produced by process p is preceded and followed by the constant 0, and the sequence of indices is terminated by the special symbol $\perp$: $0, i_1, 0,\ 0, i_2, 0,\ \ldots,\ 0, i_m, 0,\ \perp$.

To each letter l that is produced by process q, we add a flag f_u that indicates if this particular letter is the first letter of the u-word in the sequence, and a

flag f_v that indicates if it is the first letter of the v-word. Each letter is again preceded and followed by the constant 0, and the sequence is terminated by $\perp$: $0, l_1, f_{u1}, f_{v1}, 0,\ 0, l_2, f_{u2}, f_{v2}, 0,\ \ldots,\ 0, l_k, f_{uk}, f_{vk}, 0,\ \perp$.

We assume that the encoding of the indices and letters with flags have equal length N. Each encoding of $0, i, 0$ and $0, l, w_i, w_j, 0$ starts with a sequence (say, 0111) that will occur nowhere else in any sequence encoding some sequence of indices or letters with flags, which allows us to identify where the output of an index or letter with flags starts.

LTL. We set $\varphi_u = \alpha \rightarrow (\gamma \wedge (\alpha_u \rightarrow \gamma_u))$ for the following path assumptions α and α_u, and guarantees γ and γ_u:

α: Globally, the concurrent scheduling of p and q is succeeded by a sequence of $N-1$ times where only p is scheduled, which is succeeded by a finite sequence where only q is scheduled, which is succeeded by a further concurrent scheduling of p and q.

γ: Globally, the concurrent scheduling of p and q initializes the output of an index $i \in I \cup \{\perp\}$ by process p.

α_u: Globally, an output sequence of an index $i \in I \cup \{\perp\}$ that is started by a concurrent scheduling of p and q is succeeded by $|u_i| \cdot N - 1$ (where $|u_i|$ denotes the length of the word u_i) positions in which only q is scheduled, which is succeeded by a concurrent scheduling of p and q.

γ_u: The concurrent scheduling of p and q initializes the output of an index $i \in I$, followed by the output of u_i, until the concurrent scheduling of p and q initializes the emission of $\perp$ by p, followed by the emission of $\perp$ by q.

If φ_v is defined correspondingly, and $\gamma_\perp$ denotes the guarantee that $\perp$ is not immediately emitted, then $\psi = \gamma_\perp \wedge p \wedge q \wedge \varphi_u \wedge \varphi_v$ is realizable if and only if the correspondence problem has a solution.

CTL. We define a couple of auxiliary functions. First, let φ_p and φ_q be specifications, which require that the "close" output of p and q is independent of their intermediate visible input, where "close" means long enough to cover the output of a few indices and words, respectively. p and q are additionally required to output sequences of indices and words, respectively.

We then define $\varphi_u = E\,\varphi\,U\,\varphi_\perp$, where $\varphi_\perp$ denotes a formula which requires that p and q would start to emit $\perp$, and φ is the conjunction of the following assertions:

- If p would start to emit an index $i \in I \cup \{\bot\}$, and q would start to emit u_j, than $i = j$ and p and q are both scheduled concurrently.
- If q would start to emit a word u_i minus the leading 0 and p would not start to emit an index $i \in I \cup \{\bot\}$, than only p is scheduled.
- And in all other cases, only q is scheduled.

Finally, we define φ_v correspondingly, and let γ_0 denote the assertion that the output variables of p and q are set to 0. Then

$$\psi = \gamma_0 \wedge p \wedge q \wedge \varphi_p \wedge \varphi_q \wedge \varphi_u \wedge \varphi_v \wedge \neg\varphi_\bot$$

is realizable if and only if the correspondence problem has a solution.

The extension to all architectures with two or more black-box processes is straight forward:

- For the white-box processes (if any), we fix strategies that map any input history to *true*.
- For all black-box processes except p and q (if any), we specify that their output variables are globally set to *true*.
- For specifications in LTL, the existence of an arbitrary nondeterministic implementation for p and q implies the existence of deterministic implementations (cf. Subsection 9.6.5).
- And for a nondeterministic process p or q and CTL as specification language, we can specify that the implementation of p or q, respectively, is deterministic. □

The assumption $\alpha = AG \bigwedge_{A' \subseteq A} EX(\bigwedge_{p \in A'} p \wedge \bigwedge_{p \in A \setminus A'} \neg p)$ of a full scheduler can be expressed in CTL, and realizability of a CTL specification φ under full scheduling coincides with the scheduler-independent realizability of $\alpha \rightarrow \varphi$.

For LTL specifications, the undecidability of scheduler-independent synthesis follows, because realizability under full scheduling and scheduler-independent realizability coincide for trace languages: For every LTL specification φ, $\langle Y_{ct}, dir \rangle \models \varphi$ for the full scheduler implies $\langle Y_{ct}, dir \rangle \models \varphi$ for all schedulers, because the computation tree Y_{ct} is, for every scheduler, a subtree of the computation tree for the full scheduler (cf. Theorem 9.7).

Corollary 10.7 *The distributed scheduler-independent synthesis problem is undecidable for all architectures with at least two black-box processes and CTL or LTL specifications.* □

10.6 Globally Asynchronous and Locally Synchronous Systems

Putting together the results of the previous sections and the results of Chapter 6, it is only a small step to the treatment of mixed systems that are globally composed synchronously, but have local islands of synchronized processes [Gup03] (GALS systems).

A GALS system can be described as a pair $(\mathbf{A}, C)$ that consists of an architecture $\mathbf{A} = (A, B, D, \Pi, \{I_a\}_{a\in A}, \{O_a\}_{a\in A})$ and a disintegration $C = \{A_i\}_{i\in I}$ of the set $A = \biguplus_{i\in I} A_i$ of processes. To cover GALS systems, it suffices to restrict the considered schedulers to such schedulers that always schedule the processes in the same quotient at the same time (GALS scheduler). Formally, the disintegration C replaces the set of processes A in our constructions, that is, GALS scheduler are $\mathcal{O}^{gals}_{sched}$-labeled $2^{C\cup\Pi}$-trees with $\mathcal{O}^{gals}_{sched} = 2^{2^C}$, the full GALS scheduler maps every input history to 2^C, and the computation tree $\langle Y_{ct}, dir\rangle$ is a total $2^{C\cup\Pi}$-labeled $2^{C\cup\Pi}$-tree.

If all black-box processes of $\mathbf{A}$ are in the same quotient $A_b \in C$ of the disintegration, then $\mathbf{A}_b = (A_b \uplus \{env\}, B, D \cap A_b, \Pi, \{I_a\}_{a\in A_b\uplus\{env\}}, \{O_a\}_{a\in A_b\uplus\{env\}})$ with $O_{env} = \Pi \setminus \bigcup_{a\in A_b} O_a$ and $I_{env} = \Pi$ is called the *local subarchitecture* of $\mathbf{A}$.

A synthesis procedure for GALS systems can be obtained by simply composing the synthesis algorithms from Section 10.3 (for the full scheduler) or Section 10.4 (for an unknown scheduler), respectively, with the algorithm from Section 6.4 for fork-free architectures.

Theorem 10.8 *A GALS system $(\mathbf{A}, C)$ is decidable if and only if all black-box processes are in one quotient $A_b \in C$ of C and the local subarchitecture is fork-free.*

Proof: For the 'if' direction, let us first consider a simplified problem. Let $\mathbf{A}' = \mathit{feedback}(\mathbf{A}_b)$ be the feedback architecture of the local subarchitecture of $\mathbf{A}$, let, for every black-box process $b \in B$, $I'_b \supseteq I_b$ be the input to the best informed black-box process in $\mathbf{A}'$, and let $I'_w = I_w$ be the (unchanged) input to white-box processes. We consider the slightly simpler problem to construct an implementation for the GALS system $(\mathbf{A}_1, C)$ for $\mathbf{A}_1 = (A, B, D, \Pi, \{I'_a\}_{a\in A}, \{O_a\}_{a\in A})$.

Since all black-box processes are in the same quotient, they are always scheduled together, and they have access to the same input; we can therefore easily adapt the decision procedure from Section 10.3 (for full GALS scheduler) or Section 10.4 (for unknown GALS scheduler) to construct an automaton $\mathcal{D}$ that accepts all suitable implementations $P_B = \{p_b\}_{b\in B}$ of the black-box processes.

To check if $\mathcal{D}$ accepts an implementation that can be distributed (such that every process uses only its input in $\mathbf{A}'$) can then be checked by the algorithm from Subsection 6.5.2. If no such implementation exists, the original synthesis problem is not realizable. If such an implementation exists, we can construct a distributable finite state implementation and distribute it to the processes in B (with the input they have in $\mathbf{A}$), using the algorithm from Subsection 6.5.3.

For the 'only if' direction, the undecidability results for architectures that do contain an information fork from Section 6.6 carry over to systems where $\mathbf{A}_b$ contains an information fork, and the undecidability result for asynchronous systems with two or more black-box processes (Theorem 10.6) directly applies to GALS systems with black-box processes in two or more quotients. □

10.7 Conclusions

The first open synthesis algorithms for trace languages have been introduced almost simultaneously for synchronous [PR89a] and asynchronous [PR89b] systems in the late 1980's. In the synchronous paradigm, synthesis has received great attention ever since, whereas results for the asynchronous setting have been few and far between. The introduction to this chapter raised the question whether this is due to an inherent undecidability or hardness of the problem.

The results of this chapter show that the cost of synthesizing asynchronous systems depends on the treatment of the scheduler. Synthesizing asynchronous systems is computationally no more expensive than synthesizing synchronous systems, when we use the most most widespread semantics and presume a full scheduler. Asynchronous synthesis without assumptions on the scheduler, on the other hand, is exponentially harder.

The undecidability of the multi-process synthesis problem underlines that the synthesis of asynchronous systems is indeed more difficult than the synthesis of synchronous systems: While it is possible to solve the distributed synthesis problem for all fork-free architectures – which contain several architectures with multiple black-box processes like pipelines [PR90] and rings [KV01] – in the synchronous paradigm, distributed synthesis for asynchronous systems is restricted to architectures that contain only a single black-box process.

However, the solution of the distributed synthesis problem is, even when restricted to only a single black-box process, a significant step forward. Model checking (which can be viewed as the special case of the distributed synthesis problem where all processes are white-box) has brought formal methods to industrial practice in the test and verification phase. Distributed synthesis

allows the application of formal methods in a much earlier design phase. An incompletely implemented system defines an architecture with a single black-box process (representing the unfinished part of the system) in addition to the completed white-box processes. By checking the realizability of the specification for this architecture, we can recognize design errors as soon as they are introduced into the implementation.

Finally, the treatment of GALS systems connects the decidability results as well as the synthesis procedures for synchronous and asynchronous systems.

Part IV

Summary & Conclusions

Chapter 11

Summary

There are two ways to summarize this thesis: One way is to list and connect the individual results, and the second way is to outline the power and the beauty of automata-theoretic approaches. Let us abuse the lack of space restrictions to cover both perspectives.

11.1 Results

The main results of this thesis are

- an improved complexity bound and a performant strategy improvement algorithm for solving parity games,
- the decidability proof for alternating-time logics, and the determination of the complexity for the satisfiability and synthesis problem for ATμC and ATL* specifications,
- the characterization of architectures with decidable realizability and synthesis problems for synchronous, asynchronous, and GALS systems,
- the introduction of bounded synthesis, and
- the integration of probabilistic and reactive environments into our toolchain.

Parity Games. Parity games are an integral part of the automata-theoretic approaches to satisfiability checking and synthesis of Parts II and III, because testing the non-emptiness of parity tree automata reduces to solving parity games. The first part of the thesis has therefore been dedicated to the development of efficient algorithms for solving them.

In Chapter 2, we have improved the known complexity bound for solving parity games with n positions and m edges from $O(m\, n^{\lceil 0.5\, c \rceil})$ for parity games with c colors to $O(m\, n^{\gamma(c)})$ with $\gamma(c) = \frac{1}{3}c + \frac{1}{2} - \frac{1}{\lfloor 0.5\, c \rfloor \lceil 0.5\, c \rceil}$ if the number of colors is even, and $\gamma(c) = \frac{1}{3}c + \frac{1}{2} - \frac{1}{3c} - \frac{1}{\lfloor 0.5\, c \rfloor \lceil 0.5\, c \rceil}$ if it is odd.

Chapter 3 contributes a strategy improvement algorithm that can, in every step, pick the optimal combination of all local strategy improvements. This seems to be a major algorithmic breakthrough – our experimental results indicate that usually a linear part of the edges are local improvement edges, which leads to a concurrent consideration of exponentially many strategies in the size of the parity game, and thus to much faster termination.

Alternating-Time Logics. In Part II, we demonstrated the decidability and finite model property of alternating-time logics, and determined the complexity of the satisfiability and synthesis problem for ATμC (Chapter 4) as well as for ATL* (Chapter 5). The complexity results for ATL* are particularly surprising: Satisfiability checking and synthesis for ATL* is not only no more expensive than satisfiability checking and synthesis for CTL$^+$, it is also no more expensive than ATL* model checking.

While parity games are needed at the end of our automata-theoretic constructions, automata over concurrent game structures, which we introduced for the representation of ATμC specifications, are the entry point for the constructions of Part III.

Distributed Synthesis. We have shown that it is surprisingly simple to distinguish architectures with a decidable realizability and synthesis problem from those, for which realizability is undecidable. Our information fork criterion provides a simple algorithm that can check decidability for synchronous and GALS systems in time quadratic in the size of the architecture, and for asynchronous systems it even suffices to check if there is only a single black-box process.

While checking decidability is cheap, deciding realizability and, if applicable, constructing a distributed implementation is expensive: It is, even for the decidable fragment, nonelementary in the size of the specification. For the proof of decidability, we provided a uniform automata-theoretic approach, which also

allows for a simple incorporation of the automata-based synthesis techniques for probabilistic (Chapter 8) and reactive (Chapter 9) environments.

For synthesizing distributed implementations, this approach seems to be beyond price. The intuition that synthesis is therefore infeasible in practice is, however, misleading: The inherently high complexity is caused by the maximal size of a minimal solution. From a more applied point of view, too large solutions are of limited interest, because they are bound to violate implicit design constraints like the available memory. We have therefore introduced bounded synthesis, which renders the synthesis problem decidable by making this design constraint explicit, and showed that bounded synthesis is nondeterministic quasi-linear in the size of the smallest solution.

11.2 Automata-Theoretic Perspective

Reading this thesis from a technical point of view exemplifies the power and beauty of automata-theoretic constructions. For most constructions, a very basic toolset of automata transformation – comprising tools for language projection, automata dualization, nondeterminization, and intersection, a "narrowing" operation to withdraw information, and non-emptiness testing – suffices for most constructions. This basic toolset is of great service when attacking synthesis and satisfiability questions: It allows us to identify the particularities of a problem, and to approach them isolated from routine transformations.

For the satisfiability problem for ATμC specifications in Chapter 4, the particularity is the representation of the specification in a dedicated automata type – automata over concurrent game structures – and the proof of the bounded branching property for these automata.

The particularity of the satisfiability problem for ATL* specifications (Chapter 5) is the concise representation of witness strategies in a tree model.

The particularity of distributed synthesis (Chapter 6) is the order of informedness, which can be treated in isolation by applying a cheap preprocessing step, in which we check if the architecture contains an information fork, and transform a fork-free architecture into a hierarchical architecture. It is then simple to assemble an automata-based synthesis algorithm for architectures of this type from our basic toolset.

The particularities in Chapter 7 are the construction of safety automata which are, for bounded structures with a predefined bound, acceptance equivalent to parity automata.

The particularities in Chapter 8 are the construction of an acceptance game for trace languages under the assumption of ε-environments, its translation to weak alternating automata, and the utilization of the structure of these automata for efficient nondeterminization.

And, finally, the particularity in Chapter 10 is the treatment of two different kinds of incomplete information by a single automata transformation – by simulating the run of an automaton on a relaxed implementation by an automaton that reads an ordinary implementation as input.

This technical perspective allows us to appreciate the contributions of a generation of researchers that lead to the rich collection of tools available today. From this perspective, this thesis offers some modest extensions to this toolset to the scientific community: We added automata that recognize the models of alternating-time specifications, we decoupled the complexity bound for testing the emptiness of alternating automata from the branching degree, we added a technique for the concise representation of witness strategies, we can now translate acceptance under the assumption of ε-environments to standard automata, and we added a tool for the treatment of asynchronicity in various forms.

Chapter 12

Conclusions

We have identified the fundamental parameters of the distributed synthesis problem – system architecture, process cooperation, process composition, and environment model – and analyzed the synthesis problem along these dimensions.

While studying distributed synthesis from different angles provides more and more details, it also provides a general insight into the conceptual differences between decidable and undecidable classes of synthesis problems. This insight can be phrased as the *decidability rule of thumb*:

> Realizability is decidable if and only if the knowledge of the black-box processes is pairwise comparable.

System Architecture. The system architecture is the natural starting point for studying distributed synthesis. For synchronous systems, the information fork criterion is an instance of the decidability rule of thumb. In the widespread setting of deterministic processes with a dedicated nondeterministic environment, realizability and synthesis become undecidable if there are pathways for the transmission of a secret (a stream of bits that can be generated nondeterministically by the environment) to two processes such that the respective other process cannot intercept the transmission of the secret.

Process Cooperation. Traditionally, we assume a setting for open and distributed synthesis where all processes but a dedicated environment are deterministic. Allowing for alternating-time specification languages, however, requires

the consideration of nondeterministic system processes, which raises the question of the influence of this nondeterminism on the decidability of the synthesis problem.

Alternating-time specification languages cause a division of the processes by assigning them different objectives. We showed that objective driven distribution has no impact on the decidability of the satisfiability problem of temporal and fixed-point logics. This is in accordance with the decidability rule of thumb, because all processes have complete information in this setting.

The influence of nondeterminism on architecture driven distribution is indirect. While the decidability rule of thumb remains valid, the informedness hierarchy needs to be *explicit* for nondeterministic processes – that is, the set of input variables needs to be pairwise comparable – whereas it can be *implicit* for deterministic processes. Pipelines – the classic example of decidable architectures that have already been studied by Pnueli and Rosner – become undecidable if we allow for nondeterministic implementations, even if we restrict the specification language to CTL. This effect is caused by the fact that every process becomes a source of nondeterminism – and thus of secrets – in this setting.

Process Composition. For asynchronous systems, the decidability rule of thumb explains why an architecture becomes undecidable as soon as it contains two black-box processes: The incomparableness of their level of informedness is a consequence of the fact that the scheduler has means to provide them with incomparable information, even if they see the same input variables.

This power of the scheduler is restricted when we consider GALS systems: Here, it cannot interfere with the relative informedness of different processes within the same locally synchronous area. Applying the decidability rule of thumb, we see that GALS systems are decidable if and only if all black-box processes reside in a single fork-free synchronized area.

Environment Model. While the other three parameters have a significant impact on the decidability of the realizability and synthesis problems, changing the environment model from maximal to probabilistic or reactive models has no impact on the decidability of distributed synthesis – technically, the change of the environment model only effects the initial automaton for the synthesis procedure. This is in accordance with our decidability rule of thumb: The environment model has no impact on the relative informedness of different black-box processes.

Bibliography

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.

[AHM+98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998), 28 June – 2 July, Vancouver, British Columbia, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998.

[AM94] Anuchit Anuchitanukul and Zohar Manna. Realizability and synthesis of reactive modules. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV 1994), 21–23 June, Stanford, California, USA*, volume 818 of *Lecture Notes in Computer Science*, pages 156–168. Springer-Verlag, 1994.

[BC96] Girish Bhat and Rance Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996), 27–30 July, New Brunswick, New Jersey, USA*, pages 304–312. IEEE Computer Society Press, 1996.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[BCJ+97] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. Marrero. An improved algorithm for the evaluation of fixpoint expressions. *Theoretical Computer Science*, 178(1–2):237–255, 1997.

[BDHK06] Dietmar Berwanger, Anuj Dawar, Paul Hunter, and Stephan Kreutzer. Dag-width and parity games. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), 23–25 February, Marseille, France*, volume 3884 of *Lecture Notes in Computer Science*, pages 524–436. Springer-Verlag, 2006.

[BL69a] J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *Journal of Symbolic Logic*, 34(2):166–170, 1969.

[BL69b] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[Bra96] Julian C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR 1996), 26–29 August, Pisa, Italy*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246. Springer, 1996.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, 1960, Berkeley, California, USA*, pages 1–11. Stanford University Press, 1962.

[BV07] Henrik Björklund and Sergei Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Discrete Applied Mathematics*, 155(2):210–229, 2007.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings, IBM Workshop on Logics of Programs, May 1981, New York, New York, USA*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1982.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic

specifications. *Transactions On Programming Languages and Systems*, 8(2):244–263, 1986.

[CFG+01] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proceedings of 13th International Conference on Computer Aided Verification (CAV 2001), 18–22 July, Paris, France*, volume 2102 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Chu63] Alonzo Church. Logic, arithmetic and automata. In *Proceedings of the International Congress of Mathematicians, 15–22 August*, pages 23–35, Institut Mittag-Leffler, Djursholm, Sweden, 1962, Stockholm 1963.

[CMT99] Ilaria Castellani, Madhavan Mukund, and P. S. Thiagarajan. Synthesizing distributed transition systems from global specification. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999)*, pages 219–231, 1999.

[CY95] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[dA99] L. de Alfaro. From fairness to chance. *Electronic Notes in Theoretical Computer Science (ENTCS), Proceedings of the First International Workshop on Probabilistic Methods in Verification (PROBMIV 1998), 19–20 June, Indianapolis, Indiana, USA*, 1999.

[dAHM01] Luca de Alfaro, Thomas A. Henzinger, and Rupak Majumdar. From verification to control: Dynamic programs for omega-regular objectives. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), 16–19 June, Boston, Massachusetts, USA*, pages 279–290. IEEE Computer Society Press, 2001.

[dRLP98] Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference. (COMPOS 1997)*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree automata, μ-calculus and determinacy. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS 1991), 1–4 October, San Juan, Puerto Rico*, pages 368–377. IEEE Computer Society Press, October 1991.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model-checking for fragments of μ-calculus. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 1993), 8–12 July, Boulder, Colorado, USA*, volume 2725 of *Lecture Notes in Computer Science*, pages 385–396. Springer-Verlag, 1993.

[EL86] E. Allen Emerson and C. Lei. Efcient model checking in fragments of the propositional μ-calculus. In *Proceedings of the Symposium on Logic in Computer Science (LICS 1986), June 16–18, Cambridge, Massachusetts, USA*, pages 267–278. IEEE Computer Society Press, 1986.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (volume B): formal models and semantics*, pages 995–1072. MIT Press, 1990.

[EY80] Shimon Even and Yacov Yacobi. Relations among public key signature systems. Technical Report 175, Technion, Haifa, Israel, March 1980.

[FS05a] Bernd Finkbeiner and Sven Schewe. Semi-automatic distributed synthesis. In *Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005), 4–7 October, Taipei, Taiwan*, volume 3707 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2005.

[FS05b] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005), 26–29 June, Chicago, Illinois, USA*, pages 321–330. IEEE Computer Society Press, 2005.

[FS07] Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Proceedings of the 2nd Workshop for Automated Formal Methods (AFM 2007), 6 November, Atlanta, Georgia, USA*, pages 69–76. ACM Press, 2007.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of 13th International Conference on Computer Aided Verification (CAV 2001), 18–22 July, Paris, France*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 2001.

[GPFW97] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) Problem: A survey. In Ding-Zhu Du, Jun Gu, and Panos Pardalos, editors, *Satisfiability Problem: Theory and applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–152. American Mathematical Society, 1997.

[Gup03] Rajesh Gupta, editor. *First International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Architectures (FMGALS)*, September 2003.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[JPZ06] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006), 22–26 January, Miami, Florida, USA*, pages 117–123. ACM Press, 2006.

[Jur98] Marcin Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, November 1998.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2000), February, Lille, France*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer-Verlag, 2000.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC 1984), 30 April – 2 May, Washington, DC, USA*, pages 302–311. ACM Press, 1984.

[Kha79] L. G. Khachian. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[KM72] F Klee and G J Minty. How good is the simplex algorithm? *Inequalities III*, pages 159–175, 1972.

[KMTV00] Orna Kupferman, P. Madhusudan, P.S. Thiagarajan, and Moshe Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000), 22–25 August, University Park, PA, USA*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107. Springer-Verlag, 2000.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[KR03] Steve Kremer and Jean-François Raskin. A game-based verification of non-repudiation and fair exchange protocols. *Journal of Computer Security*, 11(3):399–430, 2003.

[KV95] O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995), 26-29 June, San Diego, California, USA*, 1995.

[KV97a] O. Kupferman and M.Y. Vardi. Module checking revisited. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997), 22–25 June, Haifa, Israel*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 1997.

[KV97b] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete informatio. In *Proceedings of the 2nd International Conference on Temporal Logic (ICTL 1997), 14–18 July, Manchester, UK*, pages 91–106, 1997.

[KV99] Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2):245–263, June 1999.

[KV00] Orna Kupferman and Moshe Y. Vardi. μ-calculus synthesis. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS 2000), 28 August–1*

September, Bratislava, Slovakia, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer-Verlag, 2000.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS 2001), 16–19 June, Boston, Massachusetts, USA*, pages 389–398. IEEE Computer Society Press, 2001.

[KV05] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proceedings 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October, Pittsburgh, PA, USA*, pages 531–540, 2005.

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Lan05] M. Lange. Solving parity games by a reduction to SAT. In *Proceedings of the 2nd International Workshop on Games in Design and Verification (GDV 05), 12 July, Edinburgh, UK*, 2005.

[LR81] Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages (POPL 1981), January, Williamsburg, Virginia, USA*, pages 133–138. ACM Press, 1981.

[Lud95] Walter Ludwig. A subexponential randomized algorithm for the simple stochastic game problem. *Information and Computation*, 117(1):151–155, 1995.

[Mai03] Patrick Maier. *A Lattice-Theoretic Framework For Circular Assume-Guarantee Reasoning*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, July 2003.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(5):521–530, October 1966.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001), June*, 2001.

[MS87] David E. Muller and Paul E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, 1987.

[MS95] David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1-2):69–107, 1995.

[MT01] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), 8–12 July, Heraklion, Greece*, volume 2076 of *Lecture Notes in Computer Science*, pages 396–407. Springer-Verlag, 2001.

[Obd03] J. Obdržálek. Fast μ-calculus model checking when tree-width is bounded. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003), 8–12 July, Boulder, Colorado, USA*, volume 2725 of *Lecture Notes in Computer Science*, pages 80–92. Springer-Verlag, 2003.

[Pit06] Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science (LICS 2006), 12–15 August, Seattle, Washington, USA*, pages 255–264. IEEE Computer Society, 2006.

[PR89a] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL 1989), January, Austin, Texas, USA*, pages 179–190. ACM Press, 1989.

[PR89b] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Proceeding of the 16th International Colloquium on Automata, Languages and Programming (ICALP 1989), 11–15 July, Stresa, Italy*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer-Verlag, 1989.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS 1990), 22–24 October, St. Louis, Missouri, USA*, pages 746–757. IEEE Computer Society Press, 1990.

[PT87] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing (SICOMP)*, 16(6):973–989, 1987.

[Pur95] Anuj Puri. *Theory of hybrid systems and discrete event systems*. PhD thesis, Computer Science Department, University of California, Berkeley, 1995.

[Rab69] Michael O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the American Mathematical Society*, 141:1–35, 1969.

[Rab72] Michael O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. American Mathematical Society, 1972.

[Ros92] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Sceince, Rehovot, Israel, 1992.

[Saf88] Shmuel Safra. On the complexity of ω-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS 1988), 24–26 October*, pages 319–327, White Plains, New York, USA, 1988. IEEE Computer Society Press.

[SC79] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing (SICOMP)*, 8(2):151–174, 1979.

[Sch06] Sven Schewe. Synthesis for probabilistic environments. In *Proceedings of the 4th International Symposium on Automated Technology*

for Verification and Analysis (ATVA 2006), 23–26 October, Beijing, China, volume 4218 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 2006.

[Sch07] Sven Schewe. Solving parity games in big steps. In *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007), 12–14 December, New Delhi, India*, volume 4805 of *Lecture Notes in Computer Science*, pages 449–460. Springer-Verlag, 2007.

[Sch08a] Sven Schewe. ATL* satisfiability is 2ExpTime-complete. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II (ICALP 2008), 6–13 July, Reykjavik, Iceland*, volume 5126 of *Lecture Notes in Computer Science*, pages 373–385. Springer-Verlag, 2008.

[Sch08b] Sven Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proceedings of the 17th Annual Conference of the European Association for Computer Science Logic (CSL 2008), 15–19 September, Bertinoro, Italy*, volume 5213 of *Lecture Notes in Computer Science*, pages 368–383. Springer-Verlag, 2008.

[SF06a] Sven Schewe and Bernd Finkbeiner. The alternating-time μ-calculus and automata over concurrent game structures. In *Proceedings of the 15th Annual Conference of the European Association for Computer Science Logic (CSL 2006), 25–29 September, Szeged, Hungary*, volume 4207 of *Lecture Notes in Computer Science*, pages 591–605. Springer-Verlag, 2006.

[SF06b] Sven Schewe and Bernd Finkbeiner. Synthesis of asynchronous systems. In *Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2006), 12–14 July, Venice, Italy*, volume 4407 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2006.

[SF07a] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2007.

[SF07b] Sven Schewe and Bernd Finkbeiner. Distributed synthesis for alternating-time logics. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 22–25 October, Tokyo, Japan*, volume 4762 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2007.

[SF07c] Sven Schewe and Bernd Finkbeiner. Semi-automatic distributed synthesis. *International Journal of Foundations of Computer Science*, 18(1):113–138, 2007.

[Sma83] Steve Smale. On the average number of steps of the simplex method of linear programming. *Mathematical Programming*, 27(3):241–262, October 1983.

[Tak99] Tadao Takaoka. Theory of 2-3 heaps. In *Proceedings of the 5th Annual International Conference on Computing and Combinatorics (COCOON 1999), 26–28 July, Tokyo, Japan*, volume 1627 of *Lecture Notes in Computer Science*, pages 41–50. Springer-Verlag, 1999.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *Proceedings of the 7th International Conference On Computer Aided Verification (CAV 1995), 3–5 July, Liege, Belgium*, volume 939 of *Lecture Notes in Computer Science*, pages 267–278. Springer-Verlag, 1995.

[Var98] Moshe Y. Vardi. Reasoning about the past with two-way automata. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998), 13–17 July, Aalborg, Denmark*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer-Verlag, 1998.

[vD03] Govert van Drimmelen. Satisfiability in alternating-time temporal logic. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003), 22–25 June, Ottawa, Ontario, Canada*, pages 208–217. IEEE Computer Society Press, 2003.

[VJ00] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV*

2000), 15–19 July, Chicago, Illinois, USA, volume 1855 of *Lecture Notes in Computer Science*, pages 202–215. Springer-Verlag, July 2000.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Journal of Information and Computation*, 115(1):1–37, May 1994.

[Wil99] Thomas Wilke. CTL^+ is exponentially more succinct than CTL. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1999), 13–15 December, Chennai, India*, volume 1738 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1999.

[Wil01] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bulletin of the Belgian Mathematical Society*, 8(2), May 2001.

[WLWW06] D. Walther, C. Lutz, F. Wolter, and M. Wooldridge. ATL satisfiability is indeed ExpTime-complete. *Journal of Logic and Computation*, 16(6):765–787, 2006.

[WM03] Igor Walukiewicz and Swarup Mohalik. Distributed games. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2003), 15–17 December, Bombay, Mumbai, India*, volume 2914 of *Lecture Notes in Computer Science*, pages 338–351. Springer-Verlag, 2003.

[Wol82] Pierre Wolper. *Synthesis of Communicating Processes from Temporal-Logic Specifications*. PhD thesis, Stanford University, 1982.

[Zie98] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

[ZP96] Uri Zwick and Mike S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.