Symmetric and Efficient Synthesis

Dissertation

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

von
Rüdiger Ehlers

Saarbrücken, 2013

Dekan
Prof. Dr. Mark Groves

Prüfungsausschuss

Prof. Dr. Jan Reineke	(Vorsitzender)
Prof. Bernd Finkbeiner, Ph.D.	(Gutachter)
Prof. Roderick Bloem, Ph.D.	(Gutachter)
Prof. Moshe Y. Vardi, Ph.D.	(Gutachter)
Dr. Martin Zimmermann	(akademischer Mitarbeiter)

Tag des Kolloquiums
2. Oktober 2013

Abstract

Since the formulation of the *synthesis problem for reactive systems* by Church in the 60s, research on synthesis has lead to both theoretical insights and practical approaches for automatically constructing systems from their specifications. While the first solution of the problem was given by Büchi as early as 1969, only very recently, focus has shifted towards identifying ways to exploit the structure in reactive system specifications in order to lift the scalability of synthesis to industrial-sized designs.

The recent progress in synthesis not only lead to a renewed interest in the subject, but also shed light onto the downsides of current synthesis approaches. In the original formulation of the problem, the structure of the produced solutions was not a concern. Experiments with current synthesis approaches has however shown that the computed implementations are usually very hard to understand and have little of the structure that manually constructed implementations have. Furthermore, the scalability of current synthesis approaches is still deemed to be insufficient for many industrial application scenarios, which prevents the introduction of reactive synthesis technology into industrial design flows.

In this thesis, we tackle both of these problems for reactive synthesis. To counter the insufficient structure in the solutions, we analyse the problem of *symmetric synthesis*. In this alternative synthesis problem, the aim is to compute a solution that consists of multiple copies of the same process such that the overall system satisfies the specification. Such systems have no centralised control units, and are considered to be more robust and easier to maintain. We characterise undecidable and decidable cases of the problem, and provide a synthesis algorithm for rotation-symmetric architectures, which capture many cases of practical relevance.

To improve the scalability in synthesis, we start with a simple but scalable approach to reactive synthesis that has shown its principal applicability in the field, and extend its main idea both in terms of scope and usability. We enhance its expressivity in a way that allows to synthesise robust systems, and remove its limitation to specifications of a very special form. Both improvements yield theoretical insights into the synthesis problem: we characterise which specification classes can be supported in synthesis approaches that use parity games with a fixed number of colours as the underlying computation model, and examine the properties of *universal very-weak automata*, on which we base a synthesis workflow that combines ease of specification with a low complexity of the underlying game solving step. As a side-result, we also obtain the first procedure to translate a formula in linear-time temporal logic (LTL) to a computation tree logic (CTL) formula with only universal path quantifiers, whenever possible.

The new results on symmetric and efficient reactive synthesis are complemented by an easily accessible introductory chapter to the field of reactive synthesis that can also be read in isolation.

Zusammenfassung

Trotz der Vorzüge der Synthese reaktiver Systeme gegenüber der manuellen Konstruktion solcher Systeme ist Synthese noch nicht als Teil industrieller Vorgehensmodelle etabliert. Als Hauptgrund für diese Diskrepanz gilt allgemein, dass sowohl die Qualität der synthetisierten Systeme bei Anwendung bisheriger Methoden unzureichend ist, als auch die Skalierbarkeit aktueller Syntheseverfahren der Verbesserung bedarf. Diese Dissertation behandelt beide diese Probleme der Synthese reaktiver Systeme auf breiter Front.

Zur Verbesserung der Qualität synthetisierter Systeme wird die Synthese von strukturierten Systemen betrachtet. Experimente mit aktuellen Syntheseverfahren haben gezeigt, dass die erzeugten Implementierungen oft schwer zu verstehen sind und anders als handgeschriebene Implementierungen kaum Struktur haben. Abhilfe verschafft die Beschränkung auf die Erzeugung symmetrischer Systeme, die aus mehreren Kopien des selben Prozesses bestehen, so dass das Gesamtsystem die Spezifikation erfüllt. Solche Systeme haben keine zentrale Koordinationskomponente und werden allgemein als robuster und einfacher zu warten eingestuft. In dieser Dissertation werden entscheidbare und unentscheidbare Fälle des symmetrischen Syntheseproblems identifiziert und ein Synthesealgorithmus für rotationssymmetrische Systeme beschrieben. Diese Systemklasse deckt viele praktisch relevante Architekturen ab.

Um das Problem der mangelnden Skalierbarkeit anzugehen, wird die Hauptidee des Generalised Reactivity(1) Syntheseansatzes, welcher seine praktische Anwendbarkeit bereits unter Beweis gestellt hat, aufgegriffen und sowohl bezüglich der Expressivität als auch der Benutzbarkeit vervollständigt. Die Erweiterung der Expressivität ermöglicht es, den resultierenden Ansatz für die Synthese robuster Systeme zu nutzen, während die Benutzbarkeit für industrielle Anwendungen durch die Aufhebung der Beschränkung, dass die Spezifikation eine sehr spezielle Form haben muss, erreicht wird. Beide Erweiterungen geben Einsicht in die Theorie der Synthese: Zum einen wird die Klasse der Spezifikationen, die in Syntheseansätzen verwendet werden können, die auf dem Lösen von Paritätsspielen mit einer vordefinierten Anzahl von Farben basieren, charakterisiert. Zum anderen wird Einsicht in die Eigenschaften universeller sehr schwacher Automaten gegeben. Ein Nebenprodukt der neuen Syntheseverfahren ist die erste Prozedur, um ein Ausdruck in linear-time temporal logic (LTL) in computation tree logic mit universellen Pfadquantoren (ACTL) zu übersetzen, wann immer dies möglich ist.

Die Resultate zur symmetrischen und effizienten reaktiven Synthese werden von einer didaktisch aufbereiteten Einführung in das Gebiet der reaktiven Synthese begleitet, welche auch unabhängig von den übrigen Teilen der Dissertation gelesen werden kann.

Acknowledgements

The research on which this thesis is based would not have been possible without the support of my advisor, Bernd Finkbeiner. Not only did he always have an open door for discussions, but he also gave me the freedom to choose my own directions of research, while guiding me in a way that I never lost the overview of the big picture.

I am grateful for the privilege to have been a member of Saarland University's Reactive Systems Group. Rayna Dimitrova, Klaus Dräger, Peter Faymonville, Michael Gerke, Andrey Kupriyanov, Lars Kuhtz, Hans-Jörg Peter, Markus Rabe, Christa Schäfer, and Sven Schewe supported me in many ways, including interesting discussions that led to new research insights and their company in many fun events.

The opportunity to work with many brilliant researchers during my Ph.D. studies time was a source of inspiration. Particular thanks in this context go to my co-authors, such as Robert Mattmüller, Daniel Fass, Jan Rakow, Tobe Toben, Bernd Westphal, Matthew Lewis, Paolo Marin, Daniela Moldovan, Robert Könighofer, and Georg Hofferek, just to name a few.

My thesis committee members Roderick Bloem and Moshe Y. Vardi travelled many miles to attend my defence and I can only imagine how much time has been invested by them on reviewing this thesis – thank you for this contribution!

I would also like to take the opportunity to thank the German Science foundation (DFG), which funded my work for almost five years in the scopes of the Research Training Group "Quality Guarantees for Computer Systems" and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

Finally, I would like to give special thanks to Anna, whose love and support were crucial for making this thesis come into existence. Thank you for being a part of my life!

Contents

Chapter 1

Introduction

As errors in highly safety-critical systems can have enormous consequences, the need for formal methods to ensure the correctness of computation systems is hardly ever questioned. Classically, verification techniques are used to proof a system to be error-free after it has been engineered. However, verification cannot tame the inherent difficulty of building correct-by-construction systems, and thus only provides a partial solution. As a remedy, *synthesis* has been proposed, which releases the system designer from the task to actually build a system, and only leaves her with the task to write down the specification, from which the system is automatically synthesised.

A first formulation of the synthesis problem for reactive systems was given by Alonzo Church (1962), who defined it for specifications in monadic second order logic. Implicitly or explicitly starting from this definition, several solutions to the problem were proposed in the last 40 years, leading to a plethora of methods and specification formalisms that can be used for synthesis (see, e.g., Emerson and Clarke, 1982; Rabin, 1969, 1972; Pnueli and Rosner, 1989a,b).

Despite the progress in research on synthesis and the appealingness of the general concept, its impact in practice has been limited so far. There are a couple of reasons for this mismatch. One important point in this context is the fact that the implementations synthesised should have the same quality as manually written ones, and thus should share their understandability, maintainability, and efficient implementability. There is however no guarantee that a synthesis algorithm produces an implementation that has these traits, and in fact, experience shows that current synthesis algorithms do not produce solutions with these properties. As these traits are hard to formalise, and thus also hard to take into account in synthesis algorithms, it appears most promising to impose constraints on the *structure* of the synthesised solutions.

Orthogonal to the question of structure in synthesised systems is the problem of improving *scalability* of synthesis. Due to the high complexity of the synthesis problem, many researchers have characterised the situation as hopeless. For example, for specifications in linear-time temporal logic (LTL, Pnueli, 1977), the reactive synthesis problem is known to be 2EXPTIME-complete (Pnueli and Rosner, 1989a). Despite this fact, the last years have shown a renewed interest in taming the problem from an algorithmic perspective. The main idea in this context is to make use of the fact that in practice, many specifications either have a simple form, or a simple solution. After all, the doubly-exponential time complexity for LTL synthesis is not surprising, as the specification formalism is concise enough to allow writing specifications for which all implementations must have a number of states that is doubly-exponential in the specification length. If we assume that such specifications are not fed to a synthesis algorithm under design, we can apply techniques in the algorithm that are geared towards the more common cases.

Synthesis approaches that follow these ideas can broadly be categorised into two main lines of research. The first one is concerned with improving efficiency for synthesis from full LTL, and spans a variety of techniques based on *parity game solving* (Sohail et al., 2008; Sohail and Somenzi, 2009) and *bounded synthesis* (Schewe and Finkbeiner, 2007; Finkbeiner and Schewe, 2007; Filiot et al., 2009, 2010; Bohy et al., 2012; Ehlers, 2010a, 2011a). In the second line of research, the full expressiveness of LTL is traded against the possibility to streamline the synthesis algorithm, and thus improve its scalability. While this approach restricts the set of specifications that can be handled, the high number of applications in which implementations were successfully synthesised (see, e.g., Bloem et al., 2007a,b; Cheng et al., 2012; Chinchali et al., 2012; Xu et al., 2012; Kress-Gazit et al., 2007, 2009; Ozay et al., 2011) shows that a careful choice of the specification logic can combine the efficiency and the expressivity needed for practice.

This thesis presents progress in both the *synthesis of structured systems* and techniques for extending the applicability of synthesis from algorithmically tailored specification logics.

1.1 Symmetric synthesis

Structure in synthesised systems can have many forms. In the past, solutions to synthesise explicitly represented systems with few states (Schewe and Finkbeiner, 2007; Finkbeiner and Schewe, 2007), symbolic circuits (Bloem et al., 2007a,b; Ehlers et al., 2012a), or even distributed systems (Pnueli and Rosner, 1990; Kupferman and Vardi, 2001a; Finkbeiner and Schewe, 2005) have been given. In all of these works, structural properties of parts of the solution have however been ignored. In particular, whenever possible, system designers aim at solutions that are *symmetric*, and consist of several copies of the same component.

The quest for implementing systems in a symmetric fashion is a long one in the computer science literature. For a reactive system that reads some input stream and continuously writes some output stream, this typically means that the components run in parallel and at the same time. Symmetric systems are often easier to handle than monolithic systems, as the symmetry requirement guarantees the absence of a designated coordination controller. Furthermore, symmetric systems are typically easier to maintain than other distributed systems, as they have fewer different components, and are also often easier to understand, as the components must react in way such that no central coordination authority is necessary. Finally, due to the development of verification algorithms that can take advantage of symmetry for model checking systems (Ip and Dill, 1993, 1996; Derepas and Gastin, 2001; Hendriks et al., 2003), symmetry is also a favourable property for the certification of implementations.

The *dining philosophers problem* (Hoare, 1985) is probably the most well-known scenario in which the symmetric implementability of a specification is examined. In this problem, we assume that n philosophers sit around a (round) table and there are n pieces of cutlery on the table. Each philosopher needs to acquire the pieces of cutlery to her left and to her right at the same time in order to be able to eat. The only way for all of the philosophers not to starve is thus to time-share the cutlery. If now all philosophers have to behave in the same way, this setting becomes problematic: since there is no external input that the philosophers can rely on for choosing some designated set of philosophers that eat first, in the first step they either all take two pieces of cutlery (which does not work due to the conflicts with the neighbours), one piece of cutlery (which does not suffice for eating), or no piece of cutlery at all (which does not allow for eating either). This line of reasoning also holds for the other computation steps: whenever in the respective previous computation steps, all of the choices of the philosophers have been the same, if all philosophers behave in the same way, then this means that also in the next computation step, they behave in the same way. By an inductive argument, they can thus never eat and are destined to starve.

While the dining philosophers are a setting that is mainly of theoretical interest, it is a commonly used example for showing the limitations of symmetric solutions. To circumvent these, a wide variety of works on how to break symmetry in practice has been developed. Typically, this is done by electing a *leader* in a distributed system (Fich and Ruppert, 2003; Itai and Rodeh, 1990; Frederickson and Lynch, 1987). After a leader has been chosen, it serves as the coordinator for the further operation of the system. In a sense, this line of research is quite pessimistic: the interest in symmetry breaking stems from the fact that many applications require it in order to work in a distributed fashion. However, not for all applications, this is actually true. Saving the symmetry breaking step has many advantages. The algorithms that are typically employed for symmetry breaking require the parts of the system to be able to communicate with each other, which is often not necessary in a symmetric system. Furthermore, leader election delays the availability of the overall system's main functionality and makes the implementations larger. Thus, solutions that employ symmetry breaking should be avoided, which calls for methods to analyse whether symmetry breaking is actually necessary in an application.

This thesis presents the first thorough analysis of the symmetric synthesis problem. Its key components are the identification of the driving factors for decidability and undecidability of the reactive synthesis problem in symmetric architectures, and for a large, practically relevant, class of decidable architectures, we give the first fully automatic realisability checking procedure for determining whether a distributed reactive system can be implemented in a fully symmetric (and synchronous) way, without the need to employ symmetry breaking on top of the exploitation of asymmetry in the input. In case of a positive answer, we get an implementation for the processes in the architecture. The procedure supports all architectures in which the processes cannot read each other's outputs and all processes read all inputs to the system, but permuted in a way depending on the process identifier. The problem class supported by our algorithm allows tackling a plethora of synthesis scenarios of practical relevance, which we demonstrate by the examples of a distributed traffic light controller and a packet sorter, which

are depicted in Figure 8.8 (page 89) and Figure reffig:rotationSorter (page 90). We give their formal specifications in Chapter 16.

The synthesis procedure is based on a novel *decomposition* of the symmetry requirement into an ω-regular property and a property that can be implemented by automata transformations. This main idea is actually not bound to the symmetric synthesis setting, and is likely to be useful in other settings as well. We complement the algorithm by proving the complexity-theoretic optimality of the construction.

1.2 Efficient synthesis

While even the the first solutions to the synthesis problem of reactive systems (Büchi and Landweber, 1969; Rabin, 1972) were capable of dealing with a rich class of specifications that even subsumed linear-time temporal logic (LTL) strictly, the high complexity of the synthesis problem motivated the investigation of weaker specification logics that lend themselves to efficient synthesis algorithms. By restricting our synthesis approach to only deal with such specifications, we can both reduce the theoretical complexity of the problem as well as computation times in practice. From the approaches that follow this idea (see, e.g., Alur and La Torre, 2004), *generalised reactivity(1) synthesis* (Piterman et al., 2006), which is typically abbreviated by *GR(1) synthesis*, is probably the most well-known one. It supports specifications that consist of a set of *assumptions* and a set of *guarantees*. The assumptions are meant to be used to describe the behaviour of the environment under which the system to be synthesised should work correctly. The guarantees then describe the desired properties of the system. The overall specification in this setting states that if all of the assumptions hold, then all of the guarantees must hold. To achieve a low complexity, there is only a restricted set of LTL properties that can be used as assumptions and guarantees. These consist of (1) *initialisation* constraints that explain how the system and the environment start out, (2) *basic safety constraints* that describe how the atomic propositions may evolve from one computation cycle to the next one, and (3) *basic liveness constraints* that explain which events (whose description is free of temporal operators) must happen infinitely often along a run of the system. As these types of constraints are able to capture many specifications in practice, but still allow for an efficient synthesis procedure (that is also easy to implement), the approach has found many applications (see, e.g., Kress-Gazit et al., 2007; Bloem et al., 2007b,a; Kress-Gazit et al., 2009; Wongpiromsarn et al., 2010a; Ozay et al., 2011).

It has been noted, however, that the approach has two major drawbacks: *insufficient expressivity* and *the necessity of pre-synthesis*. While many properties in practice can be translated to a combination of the three types described above, a very important class is not supported: *stability properties*. These say that eventually, some property always holds. An example for a property that witnesses the insufficient expressivity of GR(1) synthesis is that "deadlines are never missed after some finite start-up time of the system". We will see later that the missing support for stability properties also prevents the direct application of the GR(1) synthesis approach to the synthesis of *robust* systems. *Pre-synthesis* on the other hand is the process of translating a complex specification into one that can be written as a conjunction of the three simple property types stated above. For example, if we want to express that in the second computation cycle of the system, some output signal should be set, then we cannot directly do this in the GR(1) form. However, we can extend the output signal set of the system to be synthesised, and use these signals to keep track of where we are in the run of the system. In this way, the property is encodable. The pre-synthesis process can be automated, but at the expense of worsening the performance of the synthesis process. On the other hand, manual pre-synthesis is known to be cumbersome and non-trivial.

In this thesis, we solve *both* the problems of insufficient expressivity and the necessity of pre-synthesis. We first discuss how the synthesis approach can be extended to support more properties, such as stability properties. In fact, it will be shown that a further significant extension would increase the complexity of the approach, and thus in a sense, the result is an as-efficient-as-possible specification fragment for synthesis. The approach is called *generalised Rabin(1) synthesis*, as the assumptions and properties may be anything that can be represented as a deterministic Rabin automaton with one acceptance pair. We will see how this added expressivity can be used for the synthesis of *robust* systems.

Then, we present a synthesis approach that solves the pre-synthesis problem. We propose *universal very-weak automata* (UVWs) as a modelling formalism for the assumptions and guarantees in a specification. Synthesising from these can be done in EXPTIME in the size of the specification, just as for GR(1) synthesis, and the realisability problem for such specifications is also reducible to three-colour parity game solving, as we show in this thesis. Despite this fact, UVWs are strictly more expressive

than the set of properties allowed in GR(1) synthesis, and translating the assumptions and guarantees in a GR(1) specification to this automaton type is trivial. However, UVWs are not so well suited as a direct specification formalism for the user, which prevented their application for efficient synthesis approaches in the past. We solve this problem by providing a novel algorithm that translates an LTL property to this automaton type, whenever possible. On the syntactic level, we allow full LTL, with the restriction that there actually has to exist an equivalent UVW for an assumption or guarantee in order to be admissible for our new synthesis approach. As it has been shown by Maidl (2000), this class of properties is precisely the one whose satisfaction by a reactive system is expressive in computation tree logic with only universal path quantification (ACTL) and linear-time temporal logic (LTL), which motivates the name of the approach, *ACTL ∩ LTL synthesis*.

Both of the contributions of generalised Rabin(1) and ACTL ∩ LTL synthesis can be used in isolation, and the separate presentation of them in the following will underline this fact. Nevertheless, they are not strictly orthogonal. We will thus explicitly discuss their combination as well.

1.3 Related work

Reactive synthesis Reactive synthesis is a widely studied topic in the computer science literature. The original formulation of the problem is commonly attributed to Alonzo Church (1962). A first solution to Church's problem has been given quite early by Büchi and Landweber (1969), and a few years later, Rabin (1972) described a simpler construction. Rabin's approach starts by translating the specification into a deterministic Rabin word automaton, which is then translated to a Rabin tree automaton, which in turn is checked for emptiness. Rabin's solution is not concerned with linear-time temporal logic, as this logic was only defined later by Pnueli (1977). A synthesis construction for LTL was then presented by Pnueli and Rosner (1989b). The construction requires doubly-exponential time (in the length of the specification). Pnueli and Rosner (1989a) also gave a 2EXPTIME-hardness proof, which builds upon earlier work by Vardi and Stockmeyer (1985).

The high complexity of the synthesis problem has caused many researchers to consider it to be intractable. However, with the advent of *binary decision diagrams* (BDDs, Bryant, 1986; Burch et al., 1992) and *satisfiability solving* (SAT, Davis and Putnam, 1960; Davis et al., 1962; Biere et al., 2009) in the scope of *verification*, it became clear that high complexity does not mean that the situation is hopeless: using BDDs and SAT solving, many verification problems with high complexity can be solved in a fully automated manner. The key insight behind the success of BDDs and SAT solving is that the *structure* of the problems to be solved can be exploited. Often, the reasons why a system is correct are relatively simple, and clever engineering of encodings and symbolic reasoning engines can lead to tools that exploit these reasons. It is thus not surprising that similar lines of thought emerged in the synthesis domain as well. Such effort can broadly be categorised into two main lines of research.

The first one tries to exploit the structure of *solutions*. For example, by building on *universal co-Büchi automata* (Kupferman and Vardi, 2005), we can reduce the synthesis problem to a SAT or *satisfiability modulo theory* (*SMT*) problem instance (Schewe and Finkbeiner, 2007; Finkbeiner and Schewe, 2007) if we restrict the size of the implementation. This so-called *bounded synthesis* approach actually uses the conjecture that our problem has a lot of structure twice: first of all, it is based on the idea that most practical specifications have small implementations, and secondly, that the SAT or SMT solvers can make use of the regular structure of the SAT and SMT instances computed.

The second line of research aims at exploiting the structure of *specifications*. By restricting the types of specifications that can be written, we can reduce the complexity of the synthesis problem and make it more amenable to its symbolic solution (Alur and La Torre, 2004; Piterman et al., 2006). As most designs can be characterised by specifications of a simple structure, we can synthesise arbitrary systems by strengthening the specification to fall into the supported fragment. Additionally, even if we do not restrict the specification fragment, we can still make use of typical specification forms: often, a specification has the shape $(a_1 \wedge \ldots \wedge a_m) \rightarrow (g_1 \wedge \ldots \wedge g_n)$ for some assumptions $a_1, \ldots, a_m$ and guarantees $g_1, \ldots, g_n$ (which together form the so-called *sub-properties*). In the scope of full LTL synthesis, one approach to improve synthesis scalability is to deal with the safety and non-safety sub-properties separately, and combine the two only at later steps in a synthesis process (Ehlers, 2010a, 2011a). As assumptions are not always present, a few approaches focus on specifications without assumptions. However, the form of a "*typical*" specification in practice is not agreed upon in the literature. In some works, the specification is supposed to be a simple conjunction of guarantees (Filiot

et al., 2010; Kupferman et al., 2006b; Morgenstern, 2010), whereas in others, the specification is of the form "$\bigwedge$ assumptions $\rightarrow$ $\bigwedge$ guarantees" (Bloem et al., 2007a; Ehlers, 2010a; Bloem et al., 2007b, 2010a; Godhal et al., 2011).

While the efficiency of the synthesis procedures improved over the years of research, achieving progress on the *quality* of solutions has not been forgotten along the way. A common property of manually constructed system implementations is *robustness*. In the discrete world, this term describes the low sensitivity of a system against violations of the assumptions. In such cases, a system should behave in a reasonable way even if the specification does not describe in a detailed way how the system should degrade. By introducing a cost metric, we can formalise this notion and use it for synthesis. Bloem et al. (2009) proposed a quantitative approach for specifications consisting of safety assumptions and guarantees, where the robustness definition is based on the number of computation cycles in which violations of the assumptions and guarantees are witnessed. An alternative idea is provided by Bloem et al. (2010a), where the number of guarantees that hold under the violation of environment assumptions is maximised. The corresponding synthesis problem is reduced to solving generalised Streett games.

In the area of hybrid systems and control theory, work has been performed on robust synthesis where only the continuous part of the controller to be synthesised is to be made robust (Wongpiromsarn et al., 2010b), while for the discrete part, generalised reactivity(1) synthesis is used.

Closely related to robust synthesis is also the field of *fault-tolerant synthesis*. Here, fault models and fall-back specifications that need to hold in case of fault occurrences are explicitly given as input to the synthesis process. Most works in this area are concerned with adding robustness to existing systems. A notable exception is the work by Dimitrova and Finkbeiner (2009), where fault-tolerant systems are synthesised from scratch.

Distributed reactive synthesis Since the foundational works on reactive synthesis have established a general understanding of the problem and its solution, research has focused on either making synthesis faster (see, e.g., Piterman et al., 2006) or making it more useful by incorporating additional requirements for the synthesised system into the synthesis procedure. A well-known representative topic of the second of these directions is the *synthesis of distributed systems*. Here, in addition to the specification, an architecture for a distributed system is provided, and it is the task of the synthesis procedure to obtain suitable implementations for all processes in the architecture such that the overall system satisfies the specification. The result is meant to closer resemble what engineers would manually construct, and helps in deploying an implementation to the components of an actual system. The seminal work by Pnueli and Rosner (1990) shows that the synthesis problem is not decidable for all architectures, and consequently, research has focussed on identifying the decidable cases (Pnueli and Rosner, 1990; Kupferman and Vardi, 2001a; Madhusudan and Thiagarajan, 2002; Gastin et al., 2009), such as pipeline or ring architectures. The identification of decidable architectures is concluded by the work of Finkbeiner and Schewe (2005), who showed that precisely the architectures that do not have an *information fork*, i.e., a pair of processes that are incomparably informed about the overall input to the system, have a decidable distributed synthesis problem. For a more detailed overview on the field of distributed synthesis, the interested reader is referred to Schewe (2008) and Walukiewicz (2010).

Symmetric systems Due to the large number of symmetric systems in practice and their favourable properties, symmetric systems are well-studied (Fich and Ruppert, 2003; Itai and Rodeh, 1990; Johnson and Schneider, 1985; Angluin, 1980; Lehmann and Rabin, 1981). A major line of research in this context deals with the question which systems can be implemented in a symmetric fashion, and which cannot. As there are many examples for systems that cannot be implemented in this way, some authors propose *symmetry breaking* as a way to implement systems in a symmetric fashion. In any case, however, there has to be some source of non-determinism that can be used for breaking the symmetry, such as, e.g., the order of arrival of messages in an asynchronous system, or a random number generator, which is then used by the symmetry breaking algorithm. It has been shown by example of the dining philosophers that there exists no fully deterministic solution to symmetry breaking in general (Lehmann and Rabin, 1981), so such external events are necessary.

Verifying symmetric systems From an automata-theoretic point of the view, the verification and synthesis problems for reactive systems are closely related. Thus, the verification and synthesis problems for symmetric systems can easily be thought to be equally related. In terms of verification, it has been

shown that for systems consisting of many similar processes or having internal symmetries in a single process (called *automorphisms in the state space*), these symmetries can be exploited in order to speed up the verification process (Ip and Dill, 1996; Clarke et al., 1993; Sistla and Godefroid, 2004). This line of research is however only marginally relevant for symmetric synthesis, as it is concerned with how to make verification easier in the presence of symmetry. We on the other hand do not have a system description to start with in the case of synthesis, and imposing an additional restriction on the structure of the solution typically makes the problem harder, and not easier. We show in Chapter 8 that this is indeed the case.

Synthesising symmetric systems Despite the relevance of symmetric systems in the field, little work has been performed on synthesising these. To the best of the author's knowledge, there is no other work concerned with this particular problem. Only the closely related problem of *parametrized* synthesis, where a process implementation is to be found that can be instantiated an arbitrary number of times, and correctness is guaranteed for all numbers of processes (possibly above some threshold number), has been investigated.

First work in this area has been performed by Attie and Emerson (1989, 1998). Here, an implementation for a pair of processes is synthesised. This pair of processes is then instantiated for every edge in the network of processes such that every node executes the product of all processes assigned to this node for some edge. The method is shown to be correct under certain conditions, including that the composed system does not deadlock. Attie and Emerson provide sufficient conditions for checking this.

Emerson and Srinivasan (1990) describe a decision procedure for parametrised synthesis that builds on *indexed simplified computation tree logic*, a distributed version of *simplified computation tree logic* (Emerson et al., 1989). Their method produces an implementation that can be arbitrarily scaled for processes that share their memory, and compute a sufficient bound on the number of processes for specifications are unrealisable for a low number of processes. Unfortunately, the authors of that work left a proof of the correctness of their construction to an extended version of the paper, which has not been published.

Recently, Jacobs and Bloem (2012) attacked the problem from a different angle. Starting from specifications in linear-time temporal logic without the next-time operator, they examined the problem of computing processes that can be plugged together in a token ring such that for every size of the ring, the overall system satisfies the specification. Since the problem is undecidable, they propose a semi-algorithm based on the ideas of bounded synthesis (Schewe and Finkbeiner, 2007) and a theorem about the verification of distributed systems with a ring structure by Emerson and Namjoshi (2003). Their semi-algorithm effectively reduces the synthesis problem for a ring with arbitrary many processes to the same problem for a ring with up to five processes.

1.4 Structure of this thesis

This thesis can be read in many ways. Its first part is an introduction to the field of reactive system synthesis, which explains the motivations for the concepts and constructions commonly found in the literature on this topic. This part of the thesis is strictly based on previous work, and can be read in isolation.

Readers with a theoretical interest in synthesis may want to read Part 1 and Part 2 of the thesis, which comprise the introduction to reactive synthesis, and the results on symmetric synthesis, respectively.

Readers with a practical perspective onto synthesis may want to read the introduction to synthesis and Part 3 of the thesis, which deals with efficient reactive synthesis.

To obtain a comprehensive overview of all results in this thesis, the reader may want to read all parts of the thesis, where Part 2 and Part 3 may be read out of order. Part 4 describes how symmetric synthesis and the new techniques for efficient synthesis can be combined, and contains experimental results on the combination, obtained using a prototype implementation of the approaches.

1.4.1 Part I: An introduction to synthesis

In the first part of this thesis, we give an introduction to the field of synthesis and describe the important concepts, problems and solutions in this context. We do not discuss the history of the field, or give the concepts in the order of appearance in the literature. Rather, the aim is to connect the results (and folk theorems) in a way that allows to examine *why* certain classical constructions are designed in the

way they are. Furthermore, the introduction is essentially self-contained. This means in particular that the *complete set of techniques* needed for one particular synthesis approach is provided, including all automaton transformations. We provide a number of remarks that are helpful for connecting the content of this introduction to the literature on the subject.

We start in Chapter 2 by discussing how to describe systems and specifications in general on a very low level. Starting from there, we then move towards the case of systems and specifications that are representable in a finitary way, i.e., that we can describe in finite space even though the systems under concern run for an indefinite amount of time.

We then have a closer look at finitary specification formalisms. We introduce linear-time temporal logic (LTL) as a specification logic that is suitable for engineers to describe systems, and discuss ω-automata as a tool to mechanise verification and synthesis tasks. To mediate between the two concepts, we describe how to translate from LTL to an ω-automata class in Chapter 3. Along the way, we will learn about the very important concept of alternation.

Then, in Chapter 4, we have a short look at the model checking problem. This problem serves as a checkpoint in our discussion, and we deal with it to later emphasise the technical differences to the synthesis problem for those readers that are familiar with formal methods already. The subsequent introduction of games allows us to use these for our first synthesis approach that is however restricted to safety specifications. The straight-forward extension of it to more general specification classes will be shown to be sound, but incomplete.

To solve this problem, we will describe and motivate a method called *bounded synthesis* (Schewe and Finkbeiner, 2007) in Chapter 5. For its description and explanation, we will introduce tree automata. We will obtain a sound and complete synthesis methodology, and will have discussed every of its components along the way.

To obtain an understanding of other synthesis approaches and topics from the literature, we then shortly discuss some additional concepts in Chapter 6 and relate them to what we will have seen earlier. We start with parity automata and games, which are the basis for many other synthesis approaches. A discussion of the complexity of reactive synthesis will show that the bounded synthesis approach discussed in Part 1 of this thesis is actually complexity-theoretically optimal and explains why the synthesis problem has such a high complexity. Then, we take a quick glimpse at the synthesis problem from branching-time specifications, and finally discuss the synthesis of distributed systems consisting of more than one process.

In all of the chapters of the first part of this thesis, justifications for why the logics, models and concepts used traditionally in the field of synthesis make sense will be provided. However, the reasons to choose precisely these particular logics, models and concepts can often only be given by referring to the other aspects in the synthesis workflow. Thus, to counter circular references in the explanations, we delay such discussions to the very end of Part 1 of this thesis, when we end it with a summary of the concepts learned.

1.4.2 Part II: Symmetric synthesis

After the introduction to synthesis, we deal with symmetric synthesis in Part 2. We start by introducing *architectures* of symmetric systems in Chapter 8 and develop the theory to classify decidable and undecidable architectures. In Section 9, we present an algorithm to perform symmetric synthesis from linear-time specifications for rotation-symmetric architectures. A hardness proof for synthesising this class of systems shows that the construction is complexity-theoretically optimal. We then describe how the idea can be extended to the synthesis of symmetric systems from branching-time specifications, and shortly discuss how our technique can be applied to the setting in which the input arrives delayed at certain processes, which is a common case for distributed systems that are connected by some communication bus. We conclude Part 2 with a summary.

1.4.3 Part III: Efficient synthesis

The contribution of this thesis to the state of the art in efficient synthesis is based on identifying the good properties of the generalised reactivity(1) synthesis approach and incorporating these properties into new synthesis frameworks that greatly enhance its applicability. We start by revisiting generalised reactivity(1) synthesis in Chapter 11 and identify these properties.

Then, we extend the scope of this approach by describing how to incorporate stability properties. The resulting approach, called generalised Rabin(1) synthesis, is presented in Chapter 12. We show that the approach cannot be significantly extended without losing its good properties, and explain how specifications can be encoded to make use of the approach.

Then, we present ACTL $\cap$ LTL synthesis in Chapter 13, our synthesis approach that combines efficiency and ease of specification. After a discussion of the properties of universal very-weak automata, which is the driving automaton class for this approach, we show how to employ these automata in a way that is both theoretically compelling and lends itself to a symbolic implementation of the approach.

We then discuss the combination of generalised Rabin(1) and ACTL $\cap$ LTL synthesis in Chapter 14 and conclude with an outlook.

1.4.4 Part IV: Experiments & Closure

The last part of this thesis is devoted to showing the practical applicability of the concepts introduced and wrapping up the overall content. In Chapter 16, we formalise specifications for two classes of benchmarks for rotation-symmetric systems and describe the results of applying the techniques from this thesis to these specifications. Chapter 17 then gives a summary of the content of this thesis and provides an outlook.

1.5 Challenges in synthesis

Since the definition of the synthesis problem by Church (1962), many of the problems that need to be solved to bring the benefits of reactive synthesis into the practice of system development have been tackled. Yet, there are many aspects of synthesis for which the currently available conceptual and algorithmic knowledge does not yet seem to be sufficient to allow its usage in all the applications that could benefit from applying reactive synthesis algorithms.

This thesis presents progress on closing this gap. Due to the sheer number of aspects that deserve further attention, this thesis can only represent progress on some of them. Let us conclude this introduction by putting the presented concepts into the context of latest research in the area of synthesis.

1.5.1 Scalability

The problem of *scalability* of synthesis is generally considered to be the most pressing. Due to the 2EXPTIME-completeness of reactive synthesis from linear-time temporal logic, the question whether synthesis will ever scale to industrial specifications has often been raised.

Many recent works on improving scalability for full LTL synthesis focus on exploiting *compositionality*. Sohail and Somenzi (2009) presented an approach to full LTL synthesis that is mainly targeting specifications that form a big conjunction of sub-properties, most of which are safety constraints. By solving *synthesis games* for each of these constraints and pruning the games by their positions that are losing for the system player before composing the games to a synthesis game for an overall specification, they can speed up the synthesis process. A similar idea has been pursued by Filiot et al. (2010), with the improvement that multiple sub-games for non-safety specification parts are possible. Ehlers (2010a, 2012b, 2011a) considered the more general case of "$\bigwedge$ assumptions $\rightarrow$ $\bigwedge$ guarantees" specifications and presented a synthesis approach in which the specification is decomposed and that is based on binary decision diagrams (BDDs). The approach is also applicable to other specification shapes.

The other branch of research that is concerned with improving the scalability of synthesis aims at improving the *reasoning engines*, i.e., the techniques used for efficiently building a synthesis game and solving it.

Synthesis games are typically built from automata. It is generally advisable to minimise the automata before building a game in order to keep the game solving time short. Classically, methods such as *bisimulation-based quotienting* (Etessami et al., 2001) are used for this purpose. These are incomplete, but computationally cheap. Recently, Ehlers and Finkbeiner (2010) presented an approach to perform a more thorough minimisation. They describe a method that couples the use of an under-approximating automaton equivalence checker with a satisfiability (SAT) solver to search for a smaller automaton that can have a structure that is totally different than the one of the original automaton. For the case of deterministic automata, one can even apply a classical off-the-shelf SAT solver, which is guaranteed to

find an automaton of smallest possible size (Ehlers, 2010b). The resulting approach has the nice property that it complements progress on translation algorithms from LTL to automata, which is still an active field of research (see, e.g., Babiak et al., 2012; Duret-Lutz, 2011).

Another important aspect of solving synthesis games is the problem of how to efficiently reason about these. As synthesis games are huge already for simple specifications, even storing a single byte per position during game solving is beyond question. Traditionally, this problem is circumvented by using binary decision diagrams (BDDs, Bryant, 1986) as data structure for the representation of position sets in game solving, as they are not only compact, but also allow the efficient computation of the operations needed for game solving. Binary decision diagrams however have limits in compactness (Wegener, 2000) that inspired research on exchanging them for the scope of synthesis. Finkbeiner and Schewe (2007) described an approach for synthesizing systems that have small implementations using an SMT solver. Filiot et al. (2009) on the other hand exchanged BDDs by *anti-chains*. Both approaches however focus on certain scenarios, while BDD-based algorithms exist for many specification formalisms and settings. To obtain a more general replacement for BDD-based algorithms, Becker, Ehlers, Lewis, and Marin (2012) presented an algorithm for the *ALLQBF* problem, where we search for a compact Boolean-formula representation of the set of all models of a quantified Boolean formula. The resulting procedure can be used as the main oracle in a symbolic game solving process, as the universal and existential quantification in QBF formulas are suitable to reflect the two players in a synthesis game.

1.5.2 Quality of synthesised systems

Next to the issue of scalability, the other main challenge in synthesis is to ensure that the synthesised system has a *quality* that is close to the one of manually constructed systems.

A major quality criterion is the *size* of a synthesised system, which should be as small as possible. Many synthesis approaches by default construct systems that are much larger than necessary. Bloem et al. (2007b) for example considered the problem of synthesizing an on-chip bus arbiter using the generalised reactivity(1) synthesis approach (Piterman et al., 2006). Their analysis shows that using symbolic data structures for solving the synthesis problem can easily lead to the synthesised system being orders of magnitudes larger than manually made ones. Recently, Ehlers, Könighofer, and Hofferek (2012b) presented an approach to mitigate this problem by performing computational learning of Boolean functions to obtain smaller (hardware) circuits for an implementation.

Interestingly, the problem of unnecessarily large implementations has been shown not to be restricted to the case that symbolic reasoning is used for synthesis. While some synthesis approaches, such as SMT-based bounded synthesis (Schewe and Finkbeiner, 2007; Finkbeiner and Schewe, 2007) circumvent the problem by bounding the size of an implementation, in most other approaches, we build a synthesis game in which we try to find some strategy that can only visit few positions, and thus corresponds to small finite-state implementations. It was known for the case of Büchi games that finding such a "smallest" strategy is NP-hard (Clarke et al., 1995). However, as an approximate small strategy is sufficient in practice, this was not the end of the story. Recently, Ehlers (2010c) was able to show that extracting the smallest finite-state machine that represents a winning strategy in games with ω-regular winning conditions is NP-hard for any polynomial approximation quality function. This implies that for all practical purposes, finding small implementations is a hard problem. Even if we restrict our attention to *positional strategies* in the games, it can be shown that finding an approximate smallest strategy is NP-hard for any approximation factor (Ehlers, 2011c). However, from a practical perspective, finding small positional strategies has been shown to be doable for games of moderate size (Ehlers and Moldovan, 2012).

1.5.3 Impact of synthesis on system design

While the main idea of synthesis is to automate the process of system construction, it has recently been noticed that synthesis technology is also useful when manually constructing implementations.

The construction of provable correct systems typically starts by writing down its specification. At this level, first errors are introduced (Clarke and Wing, 1996; Hall, 2007). By checking the realisability of the specification or parts thereof, we can already detect contradictions in the specification at this early stage of system development. Likewise, for realisable specifications, we can synthesise a prototype and then simulate it in order to detect missing parts of the specification. These basic ideas are complemented

by more elaborate methods for specification debugging in the scope of synthesis that researchers have developed in the last years (Könighofer et al., 2009, 2010; Li et al., 2011; Raman and Kress-Gazit, 2011).

After a specification is correct and complete (enough), apart from using it for verification and synthesis purposes, it is also useful for performing *runtime monitoring* in order to detect erroneous behaviour of the system after its deployment. Recently, Ehlers and Finkbeiner (2011b,a) proposed a new approach to monitoring systems at runtime. By analysing a synthesis game for a specification, they can compute monitors for safety-critical systems that inform the user of situations that permit a strategy for the environment to falsify the specification of a system. Thus, errors in systems can be detected at runtime that *may* occur depending on the next input, but are unavoidable. Surely, such a case represents an error in a system and should be detected.

Part I

An introduction to reactive synthesis

Chapter 2

Modelling systems and specifications

Let us start this introduction to reactive system synthesis with a low-level description of what a reactive system formally is, how we can represent the behaviour of a reactive system, and how we can describe the specification of a reactive system.

To actually specify, verify, and synthesise reactive systems in a practical manner, we then restrict our attention to finitary ways of describing reactive systems and their specification.

2.1 Modelling systems – the general case

In this introduction to synthesis, we are concerned with *synchronous reactive systems*. These work in *computation cycles*. In every cycle, the system reads its input valuation and writes an output valuation. The number of computation cycles for which the system runs is not determined a-priori. If we observe and log the input and output to/from the system along its execution starting from its *initial state*, we call this a *trace* of the system.

In every computation cycle, the controller reads the values (bits) from its input *signals* and writes values to its output signals. The controller has no predefined time of going out of service, but rather must stay operational for an indefinite amount of time (until it is switched off by an external event). We will discuss at the end of this part of the thesis that assuming that the system *never* goes out of service is a feasible abstraction for both verification and synthesis. A trace of the system thus has an infinite length.

Example 1. To explain these concepts, we discuss a *coffee maker controller* example here. Figure 2.1 shows the interface of such a controller. The controller has two input bits that it reads in every computation cycle. Input bit e is supposed to represent whether the *emergency stop* button of the maker has been pressed, while c represents whether the *coffee* button has been pressed, which the user of the maker uses for signalling the wish to obtain a cup of coffee. Output bit g triggers the *grinding unit* of the coffee maker, while output bit b does the same for the *brewing unit*.

Table 2.1 shows an example of a trace of the coffee maker controller. In this trace, the coffee button is pressed in the second computation cycle. Afterwards, the grinding unit is triggered for three seconds, with a subsequent brewing process for four seconds. In the eleventh clock cycle, the user presses the emergency stop button. ★

Note that the assumption that the controller is clocked in a discrete-time way can easily be justified. First of all, most microprocessors have a fixed cycle rate and thus behave in such a way anyway. Secondly, a fixed clock rate serves as an abstraction to more complex time models. In the example above, we could assume that all clock ticks take one second. This time frame is sufficient to explain the functionality of the coffee maker and will allow us later to describe its specification.

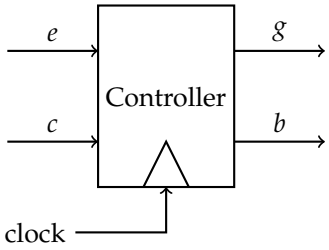

Figure 2.1: The input/output interface of a (clocked) coffee maker.

Input/Output signals	Signal valuations											
e	0	0	0	0	0	0	0	0	0	0	1	...
c	0	1	0	0	0	0	0	0	0	0	0	...
g	0	0	1	1	1	0	0	0	0	0	0	...
b	0	0	0	0	0	1	1	1	1	0	0	...

Table 2.1: A trace of a coffee maker.

Formally, a synchronous reactive system has an *input alphabet* Σ^I and an *output alphabet* Σ^O. A trace of the system is an infinite word $w = w_0 w_1 w_2 \ldots$ in which for every $i \in \mathbb{N}$, we have $w_i \in \Sigma^I \times \Sigma^O$. The element w_0 represents the input and output in the first computation cycle, w_1 represents the input and output in the second computation cycle, and so on. We will also call w an element of the set Σ^ω for $\Sigma = \Sigma^I \times \Sigma^O$ henceforth, where Σ is called the *combined alphabet* of the reactive system, and Σ^ω denotes the set of words over Σ that have infinite length.

Most reactive systems in practice are connected to their environment via binary input and output signals. By analogy to logic, we call these *atomic propositions*. If a reactive system only has binary inputs and outputs, we have $\Sigma^I = 2^{\mathsf{AP}^I}$ and $\Sigma^O = 2^{\mathsf{AP}^O}$ for the input atomic proposition set AP^I and output atomic proposition set AP^O. Note that we have a slight ambiguity of the superposition in the mathematical notation here: We use it both to decorate elements with input and output references, as well as to denote power sets, so 2^X refers to the power set of X for some set X here. However, the meaning will always be obvious from the context. As for the alphabet, we define $\mathsf{AP} = \mathsf{AP}^I \uplus \mathsf{AP}^O$ as the *combined atomic proposition set* of the system. We also call $(\mathsf{AP}^I, \mathsf{AP}^O)$ the *interface of a system*, as this tuple captures the externally accessible communication of the system with its environment.

By abuse of notation, we treat a subset X' of some set X and its characteristic function $f' : X \to \{\textbf{false}, \textbf{true}\}$ (which assigns to each element in X' the value **true** and **false** otherwise) interchangeably. Furthermore, for brevity, we use 0 and **false** interchangeably, as we do for 1 and **true**.

Example 2. Reconsider the coffee maker controller from Example 1. The controller has the input atomic proposition set $\mathsf{AP}^I = \{e, c\}$ and the output atomic proposition set $\mathsf{AP}^O = \{g, b\}$. The input alphabet is thus $\Sigma^I = 2^{\{e,c\}}$ and the output alphabet is $\Sigma^O = 2^{\{g,b\}}$. Traces of the system are elements of $(2^{\{e,c,g,b\}})^\omega$. The trace depicted in Table 2.1 can be written as

$$w = \varnothing\,\{c\}\,\{g\}\,\{g\}\,\{g\}\,\{b\}\,\{b\}\,\{b\}\,\{b\}\,\varnothing\,\{e\}\,\ldots,$$

or equivalently as

$$\begin{aligned} w \quad = \quad & \{e \mapsto 0, c \mapsto 0, g \mapsto 0, b \mapsto 0\} \\ & \{e \mapsto 0, c \mapsto 1, g \mapsto 0, b \mapsto 0\}\ \ldots \end{aligned}$$

when describing the characters in the word as characteristic functions over the combined atomic proposition set $\mathsf{AP} = \{e, c, g, b\}$ of the system. ★

Distinguishing between the input and output of a reactive system is crucial to understand its operation: the system can obtain arbitrary sequences of input characters, and thus has no control over its input, but has full control over its output. Furthermore, if the system is *deterministic*, then for every sequence of inputs, when starting from the system's initial state, there is precisely one way in which system can respond to the input sequence, and it will always respond in this way at runtime when observing this input sequence.

We will only discuss deterministic reactive systems in the following, as reactive systems in practice are usually designed to behave this way. We will see later in Chapter 5.6 that for the scope of synthesis, we can actually restrict our attention to deterministic systems without any drawback.

Since the input is not under the system's control, the behaviour of the system cannot be described by giving a single trace of the system, as a trace only describes what the system does for the particular input along that trace. For a different input sequence, the system can behave in a different way. Thus, in order to obtain a full description of the behaviour of the system, we must state for every input $w^I = w_0^I w_1^I w_2^I \ldots \in (2^{\mathsf{AP}^I})^\omega$ an output sequence $w^O = w_0^O w_1^O w_2^O \ldots \in (2^{\mathsf{AP}^O})^\omega$ such that $w = (w_0^I, w_0^O)\,(w_1^I, w_0^O)\,(w_2^I, w_2^O)\ldots$ is a trace of the system.

However, giving the full behaviour of a system by stating a mapping from input sequences to output sequences, i.e., from $(\Sigma^I)^\omega$ to $(\Sigma^O)^\omega$, would be inaccurate, as here, some output in a computation cycle can depend on the full input sequence. We call such a system behaviour *clairvoyant*. In order to rule out such behaviour by definition, we do not use a mapping from $(\Sigma^I)^\omega$ to $(\Sigma^O)^\omega$, but rather map prefix input sequences to the next output of the system, i.e., use a function $f : (\Sigma^I)^* \to \Sigma^O$. This way, the output of a system can only depend on the *past* input and not the *future* one.

When defining on which input characters the output of a reactive system can depend, it needs to be decided whether the input in the same computation cycle may be taken into account or not. In analogy to the case of finite-state reactive systems that we will discuss later, allowing this constitutes a *Mealy-type computation model*, whereas the case that the output needs to be decided on by the system without looking at the input in the current computation cycle is referred to as a *Moore-type computation model*.

In a Mealy-type computation model, given a function $f : (\Sigma^I)^* \to \Sigma^O$ that describes the behaviour of a reactive system, we thus have that $w = (w_0^I, w_0^O)(w_1^I, w_1^O)(w_2^I, w_2^O) \ldots \in \Sigma^\omega$ is a trace that corresponds to f if for every $i \in \mathbb{N}$, we have $w_i^O = f(w_0^I w_1^I \ldots w_i^I)$. Likewise, for a Moore-type computation model, given a function $f : (\Sigma^I)^* \to \Sigma^O$ that describes the behaviour of a reactive systems, a word $w = (w_0^I, w_0^O)(w_1^I, w_1^O)(w_2^I, w_2^O) \ldots \in \Sigma^\omega$ is a trace that corresponds to f if for every $i \in \mathbb{N}$, we have $w_i^O = f(w_0^I w_1^I \ldots w_{i-1}^I)$. Note that in the Mealy-type model, the value of $f(\epsilon)$, where ϵ denotes the empty word, is never used for any trace.

Example 3. Reconsider the coffee maker controller from Example 1 and 2. If the controller is supposed to have a Mealy-type computation model, the trace depicted in Table 2.1 witnesses the fact that a function f describing the complete behaviour of the system has $f(\varnothing) = \varnothing$, $f(\varnothing\{c\}) = \varnothing$, and $f(\varnothing\{c\}\varnothing) = \{g\}$. The trace does not tell us anything about $f(\{c\})$ or $f(\varnothing\varnothing)$.

Likewise, if the controller is supposed to have a Moore-type computation model, the trace depicted in Table 2.1 witnesses the fact that a function f describing the complete behaviour of the system has $f(\epsilon) = \varnothing$, $f(\varnothing) = \varnothing$, and $f(\varnothing\{c\}) = \{g\}$. Again, the trace does not tell us anything about $f(\{c\})$ or $f(\varnothing\varnothing)$. ★

The function $f : (\Sigma^I)^* \to \Sigma^O$ is typically referred to as the *labelling* of the *computation tree* $\langle (\Sigma^I)^*, f \rangle$ that represents the behaviour of a system. The first element of the $\langle \cdot, \cdot \rangle$ tree tuples is also called the set of *nodes* of the tree. Formally, we require for a tuple $\langle T, \tau \rangle$ to be called a tree that (1) T is a prefix-closed set over some *set of directions* X (i.e., $T \subseteq X^*$, $\epsilon \in T$ and for every $t \in X^*$ and $x \in X$, if $tx \in T$, then also $t \in T$), and (2) that $\tau : T \to Y$ for some set Y. Thus, for computation trees, the set of directions is the same as the input alphabet of the reactive systems that the computation tree is meant to describe, while the set of possible *node labels* represents the output alphabet of this system. While giving the set of nodes in the tree tuples seems to be redundant for trees describing the behaviour of a reactive system (as these must be able to react to any input), trees are traditionally denoted in this way as they are also used for other purposes, in which they are often not *full*, i.e., have $T = X^*$ for some set X. For example, we will use run trees of automata later in Section 3.1.

Example 4. Let us take a look at the coffee maker example again. Figure 2.2 depicts a computation tree of such a coffee maker to which the example trace in Table 2.1 corresponds in the Moore-type computation model. The location of the tree nodes encodes the prefix input ("input-so-far") that the node refers to, while the labelling of the node is given explicitly in the figure. The input alphabet of the coffee maker Σ_I (or, equivalently, the set of directions of the tree), has four elements, namely $\varnothing$, $\{c\}$, $\{e\}$, and $\{e, c\}$, and for every node, the left-most successor of a node corresponds to the input symbol $\varnothing$, the second-to-left-most successor is for the symbol $\{c\}$, and so on. The root is represented at the top of the figure.

A computation tree explains the behaviour of a reactive system along different input sequences. We can see here that after the coffee button is pressed by the user, the machine starts the grinding unit in the *next* computation cycle, unless the emergency stop button is pressed. If the coffee button is not pressed in the first computation cycle and is then pressed two times in a row, from the prefix of the execution tree visible in the figure, it seems as if the machine continues with the grinding process as if the coffee button has only been pressed in one computation cycle. However, due to the fact that the tree is capped after a few computation cycles along every input sequence, we cannot see if pressing the coffee button twice in a row results in an additional grinding/brewing cycle later. ★

From this example, we can see the main idea of using a computation tree to represent the behaviour of a system: the tree arranges the possible traces of a non-clairvoyant reactive systems in a way such that

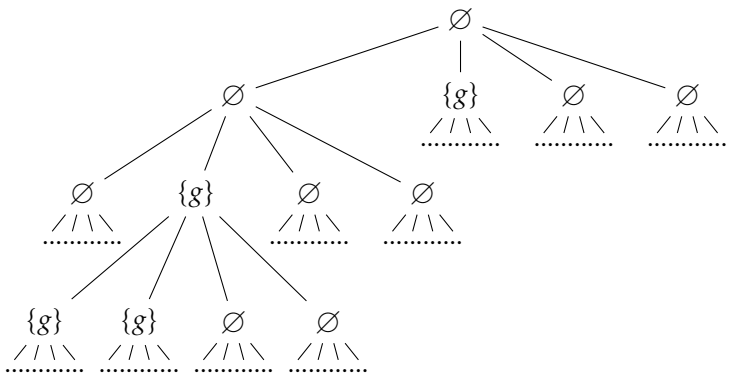

Figure 2.2: A graphical representation of the execution tree of a coffee maker.

until the input in a trace differs, we move along the same nodes and edges in the tree to read the output along the run, and thus get the same output by definition. Thus, such computation trees represent precisely the non-clairvoyant reactive systems.

We call a finite or infinite word $w^I \in (\Sigma^I)^\omega \cup (\Sigma^I)^*$ a *branch of the tree*, and say that a node $w^I \in (\Sigma^I)^*$ induces a *sub-tree* $\langle T', \tau' \rangle$ by setting $T' = \{t \in (\Sigma^I)^* \mid w^I t \in T\}$ and $\tau'(t) = \tau(w^I t)$ for every $t \in T'$

2.2 Modelling specifications – the general case

Let us now discuss how the *specification* of a reactive system can be described, which is crucial for verifying or synthesising such a system – without the specification, we do not know *what* we actually have to verify or synthesise.

Technically, desired properties of a reactive system with the interface (Σ^I, Σ^O) can be represented as a set of words in $(\Sigma_I \times \Sigma_O)^\omega$. Such a set of words is also called a *word language* or a *linear-time property*. If all traces of a reactive system are contained in the language, then the system meets the property. We call testing if all traces of a system are contained in a given language the *model checking problem*. Let us have a look at an example.

Example 5. Let $\mathsf{AP} = \{c, e, g, b\}$ be a set of atomic propositions and $\Sigma = \Sigma^I \times \Sigma^O = 2^{\{c,e\}} \times 2^{\{g,b\}}$ be the alphabet of a system. We can describe the property that a coffee machine shall always eventually grind whenever the coffee button is pressed by the following word property:

$$L = \Sigma^\omega \setminus \left(\Sigma^* \cdot \{x \in \Sigma \mid c \in x, g \notin x\} \cdot \left(2^{\{c,e,b\}}\right)^\omega \right)$$

Note that we used the commonly known regular expression operators in this example for simplicity. The trace of the coffee machine in Table 2.1 appears to be in the language, as far as we can tell from the prefix. ★

2.3 Modelling finite-state reactive systems

The abstract ways to describe reactive systems and their properties discussed so far do not help us immediately in designing algorithms for verification and synthesis. The reason is that in their full generality, computation trees and word languages are infinite and thus cannot be written down in finite space directly, which is a necessary prerequisite for any algorithmic solution of these problems.

We can circumvent this obstacle by restricting ourselves to *finite-state* systems (in this section), and *ω-regular properties* in the next section. As all reactive systems found in the field are finite-state systems, the restriction to a finite number of states is no drawback.

From a formal point of view, finite-state systems with Mealy-type semantics are typically represented as *Mealy machines*, while systems with Moore-type semantics are represented as *Moore machines*.

Definition 1. *A Mealy machine is defined as a tuple* $\mathcal{M} = (S, \Sigma^I, \Sigma^O, \delta, s_0)$ *with the set of states* S*, the input alphabet* Σ^I*, the output alphabet* Σ^O*, the transition function* $\delta : S \times \Sigma^I \to S \times \Sigma^O$ *and the initial state* s_0*. Likewise,*

a Moore machine is defined as a tuple $\mathcal{M} = (S, \Sigma^I, \Sigma^O, \delta, s_0, L)$ *with the set of states* S*, the input alphabet* Σ^I*, the output alphabet* Σ^O*, the transition function* $\delta : S \times \Sigma^I \to S$*, the initial state* s_0 *and the labelling function* $L : S \to \Sigma^O$*.*

Given an input sequence $w^I = w^I_0 w^I_1 w^I_2 \ldots$, we call $\pi = \pi_0 \pi_1 \pi_2 \ldots$ a *run of a Mealy machine* if $\pi_0 = s_0$ and for every $i \in \mathbb{N}$, we have $\delta(\pi_i, w^I_i) = (\pi_{i+1}, w^O_i)$ for some $w^O_i \in \Sigma^O$. During the run, the machine generates the trace $(w^I_0, w^O_0)(w^I_1, w^O_1)(w^I_2, w^O_2) \ldots$.

Likewise, given an input sequence $w^I = w^I_0 w^I_1 w^I_2 \ldots$, we call $\pi = \pi_0 \pi_1 \pi_2 \ldots$ a *run of a Moore machine* if $\pi_0 = s_0$ and for every $i \in \mathbb{N}$, we have $\delta(\pi_i, w^I_i) = \pi_{i+1}$. During the run, the machine generates the trace $(w^I_0, L(\pi_0))(w^I_1, L(\pi_1))(w^I_2, L(\pi_2)) \ldots$.

Thus, Mealy and Moore machines serve as *finite generators* for an infinite number of traces of infinite length. From the set of runs of a Mealy or Moore machine, we can obtain its computation tree $\langle T, f \rangle$ by applying the definition of the traces induced by a function f backwards. As the output of a finite-state machine only depends on the current state and current input, and the current state only depends on the past input to the machine, it is assured that the system is non-clairvoyant, and thus induces a unique and well-defined computation tree.

Nevertheless, let us have a look at a direct construction to obtain a computation tree from a Mealy or Moore machine description. The construction tells us how the fact that we are looking at finite-state systems manifests itself in their computation trees.

Given a Mealy machine $\mathcal{M} = (S, \Sigma^I, \Sigma^O, \delta, s_0)$, we say that $\langle T, \tau \rangle$ is an *extended computation tree* of $\mathcal{M}$ if $T = (\Sigma^I)^*$, $\tau : T \to S \times \Sigma^O$, $\tau(\epsilon) = (s_0, y)$ for some arbitrary $y \in \Sigma^O$, and for every $t \in T$ with $\tau(t) = (s, y)$ and $x \in \Sigma^I$, we have that $\tau(tx) = \delta(s, x)$.

The Moore case is analogous: given a Moore machine $\mathcal{M} = (S, \Sigma^I, \Sigma^O, \delta, s_0)$, we say that $\langle T, \tau \rangle$ is an *extended computation tree* of $\mathcal{M}$ if $T = (\Sigma^I)^*$, $\tau : T \to S \times \Sigma^O$, $\tau(\epsilon) = (s_0, L(s_0))$, and for every $t \in T$ with $\tau(t) = (s, y)$ and $x \in \Sigma^I$, we have that $\tau(tx) = (\delta(s, x), L(\delta(s, x)))$.

An extended computation tree not only describes the behaviour of a reactive system, but also denotes for every prefix input sequence in which state the Mealy or Moore automaton describing the system is in after having read the prefix input. An ordinary computation tree can be obtained by projecting the values that τ maps to onto their Σ^O parts. We can observe that for two nodes t and t' in an extended computation tree, if τ maps t and t' to the same tuple in $S \times \Sigma^O$, then the two sub-trees induced by t and t' are identical[1]. Formally, this can be proven by induction on the depth of a node in the subtree. Informally, this can be seen from the fact that the behaviour of a Mealy or Moore machine only depends on the current input/output and the current state, which are stored into the roots of the sub-trees. Thus, two sub-trees with the same root labelling have to be identical.

Example 6. Reconsider the coffee maker controller example from Chapter 2.1. Figure 2.3 contains a graphical representation of a Mealy machine for the controller. The controller starts in the state *init*, in which it waits for a key press. If the coffee button is pressed, the grinding/brewing cycle is started, with a delay of one clock cycle. If the user presses the emergency stop button, the machine moves to the state *off*, in which it remains indefinitely, and stops any brewing and grinding actions.

The part of the computation tree shown in Figure 2.2 is compatible with the Mealy machine in Figure 2.3, i.e., the computation tree could be the result of listing all runs of the Mealy machine and converting these into a computation tree, or equivalently, of computing the extended computation tree of the Mealy machine and restricting the node labelling function to the outputs.

As the Mealy machine has only 9 states and 4 different outputs, the computation tree can only have 36 distinct sub-trees. For example, as after reading $\varnothing\{c\}\varnothing$ or $\varnothing\{c\}\{c\}$ from the initial state of the Mealy machine, the machine is in the same state (namely *grin1*) and has as last output $\{g\}$ in both cases, the sub-trees from these nodes in the computation tree must be the same, meaning that the left-most and second-to-left-most sub-trees in the bottom-most level of the tree depicted in Figure 2.2 must be the same in order to be a computation tree for the Mealy machine in Figure 2.3. ★

[1] Recall that we say that a node $w^I \in (\Sigma^I)^*$ induces a *sub-tree* $\langle T', \tau' \rangle$ in a tree $\langle T, \tau \rangle$ by setting $T' = \{t \in (\Sigma^I)^* \mid w^I t \in T\}$ and $\tau'(t) = \tau(w^I t)$ for every $t \in T'$.

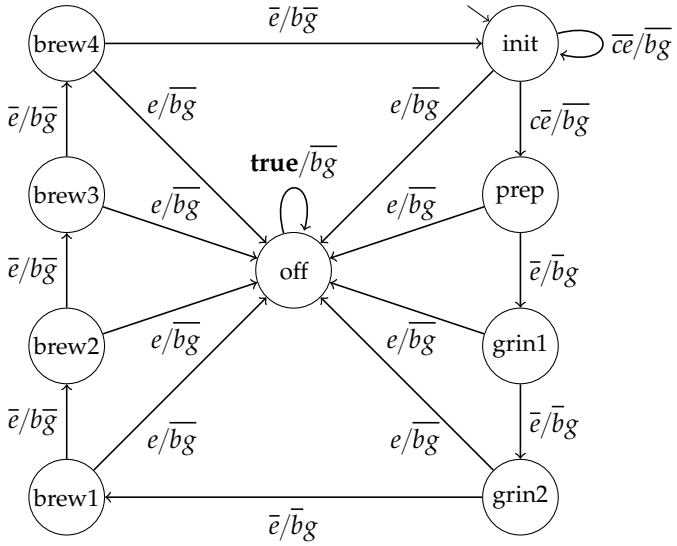

Figure 2.3: A graphical representation of a Mealy machine for a coffee maker. The initial state is marked by having an incoming edge from nowhere. Edges depict transitions between states and are labelled with the input under which they are taken and the output that is produced when taking the edge. For both purposes, we use Boolean expressions over the input and output atomic propositions, respectively, and the commonly used shortening of the expression $\neg x$ for some atomic proposition x to $\overline{x}$. Note that the Boolean expression for the input can match to more than more one element in Σ^I. For example, the expression c refers to both elements $\{c\}$ and $\{c, e\}$ of Σ^I. If we were to depict a Moore machine, we would write the labelling into the states and only label the transitions with the input under which they are taken.

2.4 Modelling the specification of a reactive system in a finitary manner

We have seen in the previous section that Mealy and Moore machines are an appropriate formalism to describe finite-state reactive systems. To verify or synthesise such systems in an automatable manner, we also need a finitary way of describing the specification of such a system.

We will discuss two formalisms for doing so in this section: formulas in *linear-time temporal logic* and *Büchi automata* over infinite words. In later chapters, we will also deal with different kinds of automata, but for the time being, Büchi automata are sufficient.

2.4.1 Linear-time temporal logic

Linear-time temporal logic (LTL) was introduced by Pnueli (1977) for the description of a system's properties. The main idea is to have formulas that may or may not hold at some position along the trace of a system. Temporal operators can be used to look into the future and quantify over the points in time to come. Thus, the temporal operators can connect the future with the current point in time. At the same time, Boolean operators can be used to connect sub-properties. For simplicity, let us introduce LTL by examples.

Example 7. Let us reconsider the coffee maker controller example. In this example, our controller has four atomic propositions as input and output signals: c, e, b and g. A trace of the system is an element of $(2^{\{c,e,b,g\}})^\omega$. An LTL formula for such a trace can use the atomic propositions c, e, b, and g to refer to these input and output signals. By default, the validity of a formula is interpreted in the first element of the trace. Thus, the LTL formula e is true for all traces in which the emergency stop button is pressed in the very first computation cycle.

LTL formulas can also contain Boolean connectives. As an example, the LTL formula $g \wedge \neg c$ is true on all traces in which the coffee button is not pressed in the first computation cycle, but the grinding unit is active in that cycle.

To look into the future, LTL has two basic *temporal operators*: X (next-time) and U (until). The former operator lets us look precisely one clock cycle into the future. For example, the formula $\mathsf{X}\, g$ holds if grinding happens in the second clock cycle. We can also nest the next-time operator - the formula $\mathsf{XX}\, b$ holds if in the third clock cycle, brewing happens. On the other hand, the binary *until operator* looks at more than one clock cycle at the same time. Given some expression $\psi\, \mathsf{U}\, \phi$ for the LTL sub-expressions ψ and ϕ, the until-expression holds if there is some clock cycle in the future in which ϕ holds, and in every clock cycle before that, ψ must hold. For example, the formula $(\neg g)\, \mathsf{U}\, c$ says that no grinding happens before at some point the coffee button is pressed. If the coffee button is never pressed, then $(\neg g)\, \mathsf{U}\, c$ does not hold. If the coffee button is pressed in the very first clock cycle, then $(\neg g)\, \mathsf{U}\, c$ is almost trivially true, as in this case, there is no clock cycle in which $\neg g$ must hold.

To simply the description of properties using LTL, there are also three temporal operators that serve as syntactic sugar: G (globally), F (finally), and R (release). The first two operators are unary, whereas the release operator is binary. For an expression ψ, $\mathsf{G}\, \psi$ holds if ψ holds in all future clock cycles, whereas $\mathsf{F}\, \psi$ holds if there is some clock cycle now or in the future in which ψ holds. For the release operator, $\psi\, \mathsf{R}\, \phi$ holds if either ϕ holds in all clock cycles, or ϕ holds in all clock cycles up to one in which ψ holds.

The G, F and R operators are technically definable as syntactic substitutions. For some LTL expressions ψ and ϕ, we have $\mathsf{F}\, \psi \equiv \mathbf{true}\, \mathsf{U}\, \psi$, $\mathsf{G}\, \psi \equiv \neg \mathsf{F} \neg \psi$, and $\psi\, \mathsf{R}\, \phi \equiv \neg(\neg \psi\, \mathsf{U}\, \neg \phi)$.

Getting back to the coffee maker, consider the LTL formula $\mathsf{F}\, b$. This formula holds (in the first clock cycle) on all traces in which at some point, brewing is performed. Using the commonly used shortcut $\psi \rightarrow \phi \equiv \neg \psi \vee \phi$ (for all sub-formulas ψ and ϕ), we are now equipped to express a property that might be part of a coffee maker controller specification: whenever at some point the coffee button is pressed, then eventually brewing should happen: $\mathsf{G}(c \rightarrow \mathsf{F}\, b)$. Note that not all traces of the controller from Figure 2.3 satisfy this property: if the user presses the emergency stop button while the machine is still grinding, then the property is violated. We can in a sense fix the property by only requiring it to apply for traces in which the emergency stop button is never pressed. While in theory, the coffee maker would then be allowed to just skip the brewing process if at some point in the future the user presses the emergency stop button, since the controller does not know this in advance, any correct controller for the specification $\mathsf{G}(\neg e) \rightarrow \mathsf{G}(c \rightarrow \mathsf{F}\, b)$ cannot make use of this fact and thus cannot expect the emergency stop button to be pressed. ★

For the sake of completeness, we now give a formal definition of the syntax and semantics of LTL.

Definition 2. *Let AP be a set of atomic propositions. For every $p \in \mathsf{AP}$, p is an LTL formula. Furthermore, we can build more complex formulas inductively. For ψ and ψ' being LTL (sub-)formulas, we have that $\neg\psi$, $\psi \vee \psi'$, $\psi \wedge \psi'$, $\mathsf{X}\, \psi$, $\mathsf{F}\, \psi$, $\mathsf{G}\, \psi$, $\psi\, \mathsf{U}\, \psi'$, and $\psi\, \mathsf{R}\, \psi'$ are also LTL formulas.*

Let $w = w_0 w_1 w_2 \in (2^{\mathsf{AP}})^\omega$ be a word. We define the validity of an LTL formula ϕ on w at some position $i \in \mathbb{N}$, written $w, i \models \phi$, inductively over the structure of w.

- *If $\phi = p$ for some $p \in \mathsf{AP}$, then $w, i \models \phi$ iff $p \in w_i$.*
- *If $\phi = \neg\psi$ for some LTL formula ψ, then $w, i \models \phi$ iff $w, i \not\models \psi$.*
- *If $\phi = \psi \vee \psi'$ for some LTL formulas ψ and ψ', then $w, i \models \phi$ iff $w, i \models \psi$ or $w, i \models \psi'$.*
- *If $\phi = \psi \wedge \psi'$ for some LTL formulas ψ and ψ', then $w, i \models \phi$ iff $w, i \models \psi$ and $w, i \models \psi'$.*
- *If $\phi = \mathsf{X}\, \psi$ for some LTL formula ψ, then $w, i \models \phi$ iff $w, i + 1 \models \psi$.*
- *If $\phi = \mathsf{F}\, \psi$ for some LTL formula ψ, then $w, i \models \phi$ iff there exists some $j \in \mathbb{N}$ such that $w, i + j \models \psi$.*
- *If $\phi = \mathsf{G}\, \psi$ for some LTL formula ψ, then $w, i \models \phi$ iff for all $j \in \mathbb{N}$, we have $w, i + j \models \psi$.*
- *If $\phi = \psi\, \mathsf{U}\, \psi'$ for some LTL formulas ψ and ψ', then $w, i \models \phi$ iff there exists some $k \in \mathbb{N}$ such that $w, i + k \models \psi'$, and for all $i \leqslant j < k$, we have $w, j \models \psi$.*
- *If $\phi = \psi\, \mathsf{R}\, \psi'$ for some LTL formulas ψ and ψ', then $w, i \models \phi$ iff either (1) there exists some $k \in \mathbb{N}$ such that $w, i + k \models \psi$, and for all $i \leqslant j \leqslant i + k$, we have $w, j \models \psi'$, or (2), for all $j \in \mathbb{N}$, we have $w, i + j \models \psi'$.*

For brevity, whenever we do not give an index on which we evaluate an LTL formula over some word, we assume an index of 0. For example, if we simply write $w \models \phi$ for some LTL formula ϕ, then we mean $w, 0 \models \phi$. If for some word w, we have $w \models \phi$, then we also say that w is a model *of ϕ. Given an LTL formula ψ, we say that some other formula ϕ is a* sub-formula *of ψ if ψ can be obtained from ϕ by applying some of the super-formula building rules stated above.*

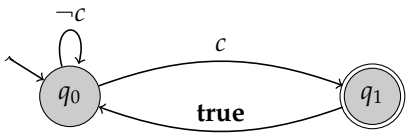

Figure 2.4: A Büchi automaton over the alphabet $\Sigma = 2^{\mathsf{AP}}$ for $\mathsf{AP} = \{c, e, b, g\}$ for the LTL property $\mathsf{GF}\, c$, using Boolean expressions over AP as edge labels.

2.4.2 Non-deterministic Büchi automata

Linear-time temporal logic is a good formalism for the designer of a system to write the specification in. Unfortunately, from an algorithmic perspective, it is a bit difficult to deal with. Consider for example the model checking problem for reactive systems. Recall from Chapter 2.2 that this means that we want to check for a given Mealy or Moore machine whether all of its runs comply to a given specification. LTL operators such as F require us to look indefinitely into the future, which is tricky, as due to the infinite number of traces of the system, we cannot list them all explicitly, and even we could, we would still be unable to write down a single trace before checking it, as even this trace is infinite. On top of that, if we were to evaluate an LTL formula from inside-out, as typically done for formulas, we would need to go back in time, as time evolves in a forward manner from the outer sub-formulas to the inner sub-formulas. Since there is no end-of-time point to start from, it is not clear how to do this.

The solution to this problem is to use *automata*. These allow us to check a reactive system against a specification by reducing this task to the problem of *trace inclusion* and permit us to do so without looking into the future of a trace, starting from the initial states of the system and the automaton. The magic then lies in the translation of the LTL formula to an automaton, but this can be done in isolation to the system under concern.

The main idea of Büchi automata is that they read a trace of a system and translate it into a run of the automaton, i.e., a sequence of states that the automaton visits when reading the trace. A run can either be accepting or not. Precisely the words that have an accepting run are accepted. Since the trace is infinite, we cannot, as we do for automata over finite words, make the acceptance of a run dependent on whether the state in which the run ends is accepting. Rather, we check for the run whether it visits accepting states *infinitely often*, and call it accepting if this is the case. We introduce Büchi automaton using graphical notation in the following example, and defer a formal description to after the example.

Example 8. Consider the Büchi automaton depicted in Figure 2.4. The accepting state q_1 is doubly-circled, and q_0 is the initial state. The automaton is deterministic, which means that for every input trace there exists only one run of the automaton.

Intuitively, the run for a word is built as follows: we start in the initial state, and then iterate over the characters in the trace. For every character, we choose a transition edge in the automaton that is labelled with the character. By writing down the states in the automaton that we observe along the way, we obtain its run. If at some point, there is no suitable transition that we can take, the run ends there and is finite even though the input word is infinite. Such a run can never be accepting.

Consider the word $w = \varnothing\{c\}\{c\}\varnothing\varnothing(\{c\})^\omega$. For the automaton in Figure 2.4, we obtain the run $\pi = q_0q_0q_1q_0q_0q_0(q_1q_0)^\omega$. As the run visits q_1 infinitely often, it is accepting and thus, the automaton from the figure *accepts the word* w. ★

We also call the set of words that is accepted by an automaton its *language*. Let us now formalise the notion of Büchi automata.

Definition 3. *A non-deterministic Büchi automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ with the finite set of states Q, the alphabet Σ, the transition function $\delta : Q \times \Sigma \to 2^Q$, the set of initial states Q_0, and the set of accepting states $\mathcal{F} \subseteq Q$.*

Given a word/trace $w = w_0w_1w_2 \ldots \in \Sigma^\omega$, we call a finite sequence $\pi = \pi_0\pi_1 \ldots \pi_k$ a run of $\mathcal{A}$ if $\pi_0 \in Q_0$, for all $0 \leqslant i < k$, we have $\pi_{i+1} \in \delta(\pi_i, w_i)$, and $\delta(\pi_k, w_k) = \varnothing$. Likewise, an infinite sequence $\pi = \pi_0\pi_1 \ldots$ is a run of $\mathcal{A}$ if $\pi_0 \in Q_0$ and for all $i \in \mathbb{N}$, we have $\pi_{i+1} \in \delta(\pi_i, w_i)$.

Given a run $\pi = \pi_0\pi_1 \ldots$, we define $\inf(\pi) = \{q \in Q \mid \forall i \in \mathbb{N}.\, \exists j > i.\, \pi_j = q\}$, i.e., $\inf(\pi)$ contains the states that occur infinitely often along the run. If $\inf(\pi) \cap \mathcal{F} \neq \varnothing$ for some run π, then we call π accepting. Given a word/trace $w \in \Sigma^\omega$, we say that $\mathcal{A}$ accepts w if there exists an accepting run for w.

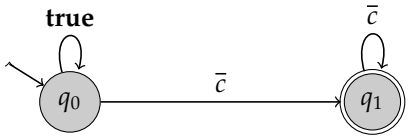

Figure 2.5: A Büchi automaton over the alphabet $\Sigma = 2^{\mathsf{AP}}$ for $\mathsf{AP} = \{c\}$ for the LTL property $\mathsf{FG}\,\neg c$, using Boolean expressions over AP as edge labels.

If $|Q_0| = 1$ and for all $(q, x) \in Q \times \Sigma$, we have $|\delta(q, x)| = 1$, then we also call $\mathcal{A}$ deterministic. The language of a state $q \in Q$ is defined to be the language of the automaton $(Q, \Sigma, \delta, q, \mathcal{F})$.

In this definition, we also introduced non-determinism for automata. Non-determinism is a fundamental concept: it suffices for a word to have *some* accepting run to be accepted, but looking at some automaton and a trace, we cannot tell with finite look-ahead which transition we have to take in order to obtain an accepting run for a word that is in the automaton's language.

Example 9. Consider the Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ with $Q = \{q_0, q_1\}$, $\Sigma = 2^{\{c\}}$, $\delta(q_0, \varnothing) = Q$, $\delta(q_0, \{c\}) = \{q_0\}$, $\delta(q_1, \varnothing) = \{q_1\}$, $\delta(q_1, \{c\}) = \varnothing$, $Q_0 = \{q_0\}$, and $\mathcal{F} = \{q_1\}$. The Büchi automaton is also depicted in Figure 2.5.

Given a word that is accepted by the automaton, the word has to end with $\varnothing^\omega$. For a corresponding accepting run, we must stay in q_0 in the run until we have reached the $\varnothing^\omega$ suffix of the trace. We cannot do so on-the-fly while observing the word with finite look-ahead, as we can never be sure if another $\{c\}$ character is still to come if we are not clairvoyant and already know when the suffix is reached. ★

Note that non-determinism is an important property in this context. In fact, it has been proven that for every LTL formula, there exists some non-deterministic Büchi automaton such that the set of models of the LTL formula coincides with the language of the automaton. However, this cannot be said for deterministic Büchi automata. It can be shown by a pumping argument that for the LTL formula $\mathsf{FG}\,a$ over the atomic propositions $\{a\}$, there exists no equivalent deterministic Büchi automaton (see Baier and Katoen, 2008, page 190 for details). The next chapter deals with the translation of an LTL formula into a non-deterministic Büchi automaton.

Chapter 3

Building non-deterministic Büchi automata from LTL formulas

In the previous chapter, we discussed the terminology to describe reactive systems and their specification along with suitable models for their finitary representation. For the systems, most definitions were simple, such as how to translate a Mealy or Moore machine to a computation tree. For the specifications, while the general approach to model these as sets of allowed traces of the system was even simpler, we discussed two finitary ways to describe the specification, namely linear-time temporal logic (LTL) as a good formalism for the manual description of properties, and non-deterministic Büchi automata as a model that was promised to be useful for actual model checking and synthesis algorithms. Before we can work our way through the latter in the next chapter, we need to mediate between the two specification formalisms. This is typically done by providing a translation from LTL to Büchi automata, which we will do in this chapter.

In the literature, there are two main approaches to accomplish this. The first one builds upon tableaux-based constructions. The second approach works using a de-tour through *alternating Büchi automata*. We discuss the latter here, as it is simpler and allows us the discuss the important concept of *alternation* along the way.

3.1 Alternating Büchi automata

The formal description of alternating automata is a bit complicated. Thus, we will start by motivating them and explain the core concepts.The main idea of alternating automata is to have a very expressive formalism that allows us to stitch together sub-automata in a rich and extremely flexible way. We do not care about the fact that the model checking or synthesis problems become more complicated for such a rich formalism as we can "compile down" an alternating Büchi automaton to a non-deterministic Büchi automaton before model checking or synthesis. At the same time, alternating automata are expressive enough to allow for a direct, almost trivial, conversion of LTL formulas into this formalism.

Let us consider the language $L = \{vv\Sigma^\omega \mid v \in \Sigma^k\} \cup \{w(\varnothing)^\omega \mid w \in \Sigma^*\}$ for some $k \in \mathbb{N}$ and $\Sigma = 2^{\mathsf{AP}}$ for some set AP. We can describe the language by some LTL formula of the following form (i.e., the set of models of the LTL formula is precisely L):

$$\left(\bigwedge_{i\in\{0,\dots,k\}} \mathsf{X}^i \left(\bigwedge_{p\in\mathsf{AP}} (p \leftrightarrow \mathsf{X}^k p)\right)\right) \vee \left(\mathsf{FG} \bigwedge_{p\in\mathsf{AP}} \neg p\right)$$

The length of the LTL formula is linear in k and linear in $|\mathsf{AP}|$.

Now assume that we want to obtain a non-deterministic Büchi automaton for L. Note that the number of states in the automaton will need to be at least $|\Sigma|^k$, and thus is at least exponential in k. To see this, recall that a word is accepted by a non-deterministic Büchi automaton if there exists at least *one* run for the word that is accepting. Along such a run, we would have to check after reading $v \in \Sigma^k$ that v is read next or eventually we only read $\varnothing$s any more in order not to have the automaton accept false-positive words. Since in the automaton, the only memory we have are the states, we would need to encode the $|\Sigma|^k$ possibilities into the states and thus have more than $|\Sigma|^k$ states.

Of course, we could simply complement the language and encode the result into a non-deterministic Büchi automaton. In fact, here, we would only need a number of states that is polynomial in k and $|\mathsf{AP}|$ when doing so. Nevertheless, we might want to avoid doing so (as we might have no use for the complement) and want a succinct formalism that can express both L and $\Sigma^\omega \backslash L$ in an efficient way.

This is precisely what alternating automata allow us to do. Non-deterministic Büchi automata can be called *automata with disjunctive branching*, as for a word to be accepted, there has to exist *one* accepting run for it, where in every step of building the run from the word, we always have to take *one* successor from the edge relation, and it suffices if there is only one. For alternating automata, this idea is generalised: in addition to disjunctive branching, we have *universal branching*, i.e., for some transitions, more than one successor from the transition function might need to be taken, requiring accepting runs in both cases.

For an alternating automaton, it then no longer suffices to look at one accepting run in isolation as a witness that a word is accepted. Rather, we need to look at several runs at one, as whenever we encounter universal branching, we need to split the run and pursue it into both directions at the same time. The runs are put together in a *run tree*, and we require that every branch in the tree is accepting.

We discuss run trees using an example. Figure 3.1 shows an alternating Büchi automaton for L with $\mathsf{AP} = \{a\}$ and $k = 2$. The automaton has some states in which the outgoing edges have classical, non-deterministic branching, and the states q_0 and q_1 have universally branching outgoing edges. Additionally, there are some transitions that lead to **true** and some transitions that lead to **false**. If a run enters the state **true**, then we say that the run is accepting by definition. Likewise, runs that eventually reach the state **false** are never accepting.

Assume that we read the word $w = \{a \mapsto 0\}\{a \mapsto 1\}\{a \mapsto 0\}\{a \mapsto 1\}(\{a \mapsto 0\})^\omega$. We start in state q_0. As the first character in the word is $\{a \mapsto 0\}$, and the transition for $\{a \mapsto 0\}$ from state q_0 has universal branching to the states q_1 and q_2, we split up the run and move to the states q_1 and q_2 simultaneously. The next character is a $\{a \mapsto 1\}$. For the run in state q_1, we move universally to states **true** and q_4. For the run in state q_2, we move forward to state q_3 by the **true**-transition. So after reading two characters from the word, we are already in three states in the automaton along different runs. The runs continue in the same manner. The runs of the automaton along the word are summed up in a *run tree*, as shown for w in Figure 3.2. Note that as all runs end in the **true** state, all runs are accepting. Thus, the whole run tree is accepting, and since thus the word has an accepting run tree, the word is accepted by the alternating automaton.

Now consider the word $w = \{a \mapsto 1\}\{a \mapsto 0\}\{a \mapsto 0\}\{a \mapsto 1\}(\{a \mapsto 1\})^\omega$. Figure 3.3 depicts a run tree for this word and the automaton from Figure 3.1. Here, the state q_- is entered after reading a few characters in two branches of the run tree, and never left. Since q_- is not an accepting state, we have two infinite branches on which accepting states are only visited finitely often and thus, the run tree is not accepting. As the alternating automaton dealt with here also has non-determinism, this does not immediately mean that the word is rejected - it would suffice if there exists *some* accepting run tree. However, for this word, this is not the case, as the only non-determinism in the automaton is in state q_-, where we can either take a transition to q_- or q_6 for any input character, but as the word ends with $\{a \mapsto 1\}^\omega$, we would need to take a transition to **false** from q_6, which would make the branch in the run tree non-accepting again. Figure 3.4 depicts such a run tree.

We now turn towards a formalisation of the ideas described above. Alternating automata are defined as 5-tuples $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$, and are thus just like non-deterministic Büchi automata, with two differences: (1) we require them to have precisely one initial state, and (2) the transition function is defined differently and of the form $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$, where $\mathcal{B}^+(Q)$ represents the set of positive Boolean functions using Q as propositions, i.e., functions that maps from 2^Q to $\mathbb{B}$ and that do not use negation. The requirements of having only one initial state and not having negation in δ are for technical convenience: they simplify the run tree definition. Automata with multiple initial states, for which it suffices to have an accepting run tree starting from any of the initial states to accept an input word, can be translated to automata with only one initial state by introducing a new state as the initial one, and taking the disjunction of the outgoing transitions of the previous initial states as transition function (for all input characters). Similar lines of reasoning also work for other automaton models, which we will make use of whenever appropriate in the following, as this simplifies the presentation. Having no negations in δ on the other hand does not restrict the expressivity of the automata, and they are not needed for many purposes anyway, including translation from LTL.

Using Boolean functions in δ, we can combine non-deterministic and universal branching in a simple and yet expressive way. For example, for the automaton from Figure 3.1, we have $\delta(q_-, \{a\}) = q_- \vee q_6$ to denote that either q_- or q_6 need to be successors of q_- in a valid run tree, and $\delta(q_0, \{a\}) = q_1 \wedge q_2$ represents that q_1 *and* q_2 need to be successors of q_0 when observing $\{a\}$ and being in q_0. The possibility to have **false** and **true** as Boolean functions allows us to express that no successor combination is allowed (and thus a run is invalid) or that we do not need any successor, respectively. This way, the special states **true** and **false** that we used in Figure 3.1 do not manifest themselves as actual states in the formal

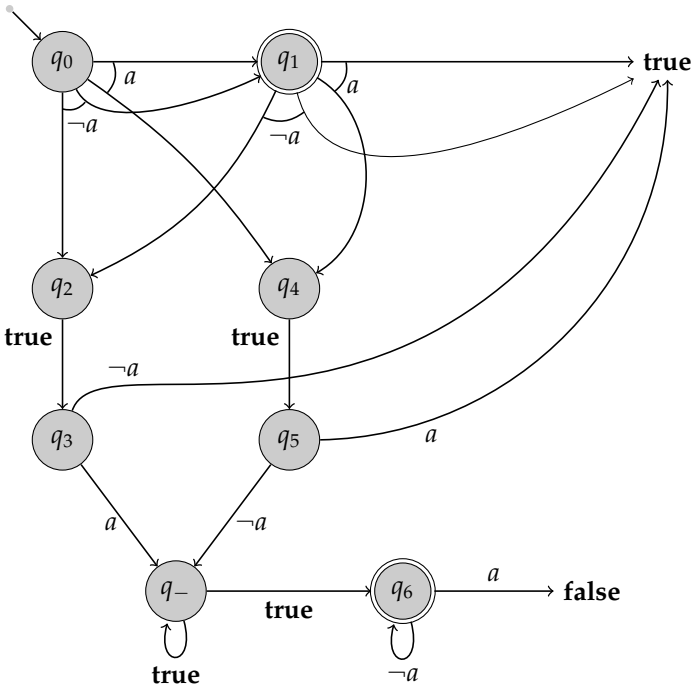

Figure 3.1: An alternating automaton over the alphabet $\Sigma = 2^{\{a\}}$, using Boolean constraints over $\mathsf{AP} = \{a\}$ to state the characters that the edges apply to. Conjunctive transitions are marked by an arc.

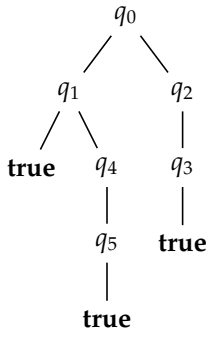

Figure 3.2: A run tree for the word $w = \{a \mapsto 0\}\{a \mapsto 1\}\{a \mapsto 0\}\{a \mapsto 1\}(\{a \mapsto 0\})^\omega$ for the alternating automaton in Figure 3.1.

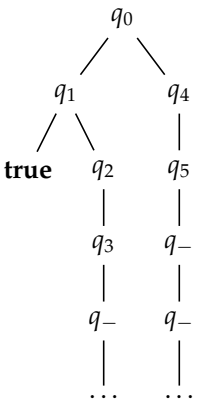

Figure 3.3: A run tree for the word $w = \{a \mapsto 1\}\{a \mapsto 0\}\{a \mapsto 0\}\{a \mapsto 1\}(\{a \mapsto 1\})^\omega$ for the alternating automaton in Figure 3.1.

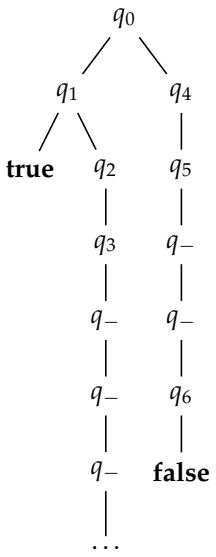

Figure 3.4: Another run tree for the word $w = \{a \mapsto 1\}\{a \mapsto 0\}\{a \mapsto 0\}\{a \mapsto 1\}(\{a \mapsto 1\})^\omega$ for the alternating automaton in Figure 3.1.

definition of alternating automata. We continue with a formal definition of run trees of alternating automata.

Definition 4. *Given an alternating Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ and a word $w = w_0 w_1 w_2 \ldots \in \Sigma^\omega$, we say that a tree $\langle T_r, \tau_r \rangle$ is a run tree of $\mathcal{A}$ and w if all of the following conditions hold:*

- $T_r \subseteq \mathbb{N}^*$, $\tau_r : T_r \to Q$
- $\epsilon \in T_r$ *and* $\tau_r(\epsilon) = q_0$
- *For every $t \in T_r$, let $S = \{tx \in T_r \mid x \in \mathbb{N}\}$ be the set of t's successor in the run tree, and $R = \{\tau(t') \mid t' \in S\}$ be the set of its labels. We translate R to a characteristic function $f : Q \to \mathbb{B}$ over Q by setting $f(q) = 1$ if $q \in R$ and $f(q) = 0$ if $q \notin R$. For the tree to be valid, we require that f satisfies the transition function δ from $\tau_r(t)$ for the character to be read next, i.e., $f \models \delta(\tau_r(t), w_{|t|})$.*

Note that in this definition, we used the same tree definition as for modelling the behaviour of reactive systems in Chapter 2.1. However, this time, we actually make use of the fact that for a tree $\langle T, \tau \rangle$, we do not necessarily need to have $T = X^*$ for some set if directions X, but rather only require $T \subseteq X^*$ with T being prefix-closed.

The definition ensures that for a valid run tree $\langle T_r, \tau_r \rangle$ in which a branch t is finite, the branch can only be finite if $\delta(\tau_r(t), w_{|t|-1}) =$ **true**. On the other hand, if along a branch in the run tree, we experience that the transition function maps the respective current combination of state and input symbol to **false**, then the run tree is in fact not valid as the definition forbids this.

Note that the definition of the run tree does not forbid us to introduce superfluous branches. For example, in the run tree in Figure 3.4, we could have q_5 as an additional successor of the root node, and **true** as an additional successor to the newly introduced q_5-node. The run tree would still be accepting, and would still be valid.

One disclaimer needs to be added however: for the example run trees in Figures 3.2 to 3.4, the **true** and **false** nodes were only added for illustration. Actual run trees do not have them, as such nodes are not needed by Definition 4.

3.2 Translating from LTL to alternating automata

After we have defined alternating automata and their run trees, for completing the picture on the transition of LTL formulas to non-deterministic Büchi automata, we now show how we can translate

an LTL formula to an alternating automaton, and how to translate the alternating Büchi automaton into a non-deterministic Büchi automaton. We explain the former here, and defer the latter to the next section. By applying both constructions in sequence, we obtain a full procedure to translate from LTL to non-deterministic Büchi automata.

The main idea behind the first construction is to first translate the LTL formula to *negation normal form*, and then use the set of sub-formulas of the LTL formula as the state set of the alternating automaton. With this idea, the definition of the transition relation and the other details of the alternating automaton are comparably straight-forward.

In the negation normal form of an LTL formula, the negation operator can only occur directly in front of an atomic proposition. Every LTL formula has another equivalent formula in this form as we made sure that our temporal operators all have a dual form. The presence of dual forms allows us to push a negation inwards. This way, we can rewrite, for example, a (sub-)formula of the form $\neg \mathsf{G}\,\psi$ for some LTL sub-formula ψ to $\mathsf{F}\,\neg\psi$. Likewise, we can rewrite temporal logic formulas of the form $\neg \mathsf{F}\,\psi$ to $\mathsf{G}\,\neg\psi$, $\neg(\phi\,\mathsf{U}\,\psi)$ to $(\neg\phi)\,\mathsf{R}\,(\neg\psi)$, and $\neg(\phi\,\mathsf{R}\,\psi)$ to $(\neg\phi)\,\mathsf{U}\,(\neg\psi)$. Together with De Morgan's laws, which allow rewriting $\neg(\psi \vee \phi)$ to $(\neg\psi) \wedge (\neg\phi)$, and $\neg(\psi \wedge \phi)$ to $(\neg\psi) \vee (\neg\phi)$, we have a complete rule system to push all occurrences of the negation operator in the formula inwards.

Example 10. Consider the LTL formula $\neg((\mathsf{G}\,p)\,\mathsf{U}\,\neg q) \vee \neg(\mathsf{F}\neg(p\,\mathsf{U}\,q))$ for the atomic proposition set $\{p, q\}$. We can rewrite this formula into negation normal form as follows:

$$\begin{aligned} & \neg((\mathsf{G}\,p)\,\mathsf{U}\,\neg q) \vee \neg(\mathsf{F}\neg(p\,\mathsf{U}\,q)) \\ \equiv\ & ((\neg\mathsf{G}\,p)\,\mathsf{R}\,\neg\neg q) \vee \mathsf{G}(\neg\neg(p\,\mathsf{U}\,q)) \\ \equiv\ & ((\mathsf{F}\,\neg p)\,\mathsf{R}\,q) \vee \mathsf{G}(p\,\mathsf{U}\,q) \end{aligned}$$

★

Note that the *size of the LTL formula*, i.e., its number of operators and occurrences of atomic propositions, does not change much in the course of pushing the negations inwards. In fact, every pushing-in operation can introduce at most one additional negation, and the number of steps to perform when translating a formula to NNF is bounded by the number of operators present in the formula. Thus, the size of the formula cannot grow by more than a factor of 2.

After the discussion on translating LTL formulas into negation normal form, we now turn towards finally translating it into an alternating Büchi automaton. The high expressivity of alternating automata allows us to encode every sub-formula into only one state, using transitions to the states of the sub-sub-formulas. When translating a formula to negation normal form, we have removed all negations from elements that are not atomic propositions. Intuitively, we have done so because the Boolean functions that the transition function of an alternating automaton maps to are required to be positive.

Let us assume that all sub-formulas in an LTL formula have one state in the automaton whose language is the set of models of the respective sub-formula. For example, an alternating automaton for the LTL formula $p \vee (\mathsf{G}\,q)$ would have the states q_p, q_q, $q_{\mathsf{G}\,q}$, and $q_{p \vee (\mathsf{G}\,q)}$. Now consider a state of the form $q_{\psi \vee \phi}$ for some sub-formulas ψ and ϕ, and let us discuss how the transition function for this state would look like. From a state labelled by $\psi \vee \phi$, we need to accept all words that either satisfy ψ or ϕ. Thus, in order to capture the meaning of $\phi \vee \phi$, we would set $\delta(q_{\phi \vee \psi}, x) = \delta(q_\phi, x) \vee \delta(q_\psi, x)$ for all $x \in \Sigma$ - alternating automata allow us to do so. For a state $q_{\phi \wedge \psi}$, we would define the transition function in an analogous way and set $\delta(q_{\phi \wedge \psi}, x) = \delta(q_\phi, x) \wedge \delta(q_\psi, x)$ for all $x \in \Sigma$, using the good expressive power of alternating automata again. A more complex example is $\mathsf{G}\phi$ for some sub-formula ϕ. Here, we need to check that ϕ holds for all future computation cycles. We can achieve this by setting $\delta(q_{\mathsf{G}\,\phi}, x) = q_{\mathsf{G}\,\phi} \wedge \delta(q_\phi, x)$ for all $x \in \Sigma$. This way, a valid computation tree for a word would need to have a branch that stays in $q_{\mathsf{G}\,\phi}$ once it enters the state, but at the same time would also need to branch to q_ϕ in every cycle, which is precisely what we need for checking $\mathsf{G}\phi$.

If we build the transition relation for our alternating automaton in a bottom-up fashion by starting with the smallest sub-formulas, and iteratively building it for the states corresponding to larger and larger sub-formulas, the transition function of the automaton is guaranteed to not have any circular definitions, and is thus well-defined. By giving bottom-up automaton building rules like in the previous paragraph for all LTL operators (and atomic propositions and their negation), and choosing the accepting states for the automaton in a meaningful way, we obtain a full translation procedure. For the latter, note that for states of the form $q_{\phi \wedge \psi}$, if we ensure that there is never a transition from a state q_f to a state $q_{f'}$ if f is a sub-formula of f', it does not matter if we make $q_{\phi \wedge \psi}$ an accepting state or not, as it can only

occur once along every branch in a run tree for the alternating automaton. The only states that might occur infinitely often along a branch are of the forms $q_{\mathsf{G}\,\psi}$, $q_{\mathsf{F}\,\psi}$, $q_{\psi\,\mathsf{U}\,\phi}$ and $q_{\psi\,\mathsf{R}\,\phi}$. For the first of these, we have already discussed how the transition functions looks like for such states above - once such a state is entered, there is a branch in which the state is never left. For a run tree to have a chance of being accepting then, we will need to make $q_{\mathsf{G}\,\psi}$ accepting. Otherwise, the language of $q_{\mathsf{G}\,\psi}$ would be $\varnothing$.

We now give a full definition of the construction to translate an LTL formula in negation normal form to an alternating Büchi automaton. Afterwards, we discuss a small example.

Definition 5. *Let ψ be an LTL formula in negation normal form over the atomic proposition set AP. We define the alternating automaton $\mathcal{A}_\psi$ for ψ as $\mathcal{A}_\psi = (Q, \Sigma, \delta, q_0, \mathcal{F})$ with:*

- $Q = \{q_\phi \mid \phi \text{ is a sub-formula of } \psi\}$
- $\Sigma = 2^{\mathsf{AP}}$
- *For all states of the form q_p for $p \in AP$ in Q and all $x \in \Sigma$, $\delta(q_p, x) =$ **true** if $p \in x$, and $\delta(q_p, x) =$ **false** otherwise.*
- *For all states of the form $q_{\neg p}$ for $p \in AP$ in Q and all $x \in \Sigma$, $\delta(q_{\neg p}, x) =$ **false** if $p \in x$, and $\delta(q_{\neg p}, x) =$ **true** otherwise.*
- *For all states of the form $q_{\phi \vee \phi'}$ for some sub-formulas ϕ and ϕ' and all $x \in \Sigma$, $\delta(q_{\phi \vee \phi'}, x) = \delta(q_\phi, x) \vee \delta(q_{\phi'}, x)$.*
- *For all states of the form $q_{\psi \wedge \phi}$ for some sub-formulas ψ and ϕ and all $x \in \Sigma$, $\delta(q_{\phi \wedge \phi'}, x) = \delta(q_\phi, x) \wedge \delta(q_{\phi'}, x)$.*
- *For all states of the form $q_{\mathsf{X}\,\phi}$ for some sub-formula ϕ and all $x \in \Sigma$, we have $\delta(q_{\mathsf{X}\,\phi}, x) = q_\phi$.*
- *For all states of the form $q_{\mathsf{G}\,\phi}$ for some sub-formula ϕ and all $x \in \Sigma$, we have $\delta(q_{\mathsf{G}\,\phi}, x) = q_{\mathsf{G}\,\phi} \wedge \delta(q_\phi, x)$.*
- *For all states of the form $q_{\mathsf{F}\,\phi}$ for some sub-formula ϕ and all $x \in \Sigma$, we have $\delta(q_{\mathsf{F}\,\phi}, x) = q_{\mathsf{F}\,\phi} \vee \delta(q_\phi, x)$.*
- *For all states of the form $q_{\phi\,\mathsf{U}\,\phi'}$ for some sub-formulas ϕ and ϕ' and all $x \in \Sigma$, we have $\delta(q_{\phi\,\mathsf{U}\,\phi'}, x) = \delta(q_{\phi'}, x) \vee (q_{\phi\,\mathsf{U}\,\phi'} \wedge \delta(q_\phi, x))$.*
- *For all states of the form $q_{\phi\,\mathsf{R}\,\phi'}$ for some sub-formulas ϕ and ϕ' and all $x \in \Sigma$, we have $\delta(q_{\phi\,\mathsf{R}\,\phi'}, x) = \delta(q_{\phi'}, x) \wedge (q_{\phi\,\mathsf{R}\,\phi'} \vee \delta(q_\phi, x))$.*
- $q_0 = q_\psi$
- $\mathcal{F} = \{q_{\mathsf{G}\,\phi} \mid \phi \text{ is a sub-formula of } \psi\} \cup \{q_{\phi\,\mathsf{R}\,\phi'} \mid \phi \text{ and } \phi' \text{ are sub-formulas of } \psi\}$

Theorem 1. *Let ψ be an LTL formula over some set of atomic propositions AP, and $\mathcal{A}_\psi$ be the alternating Büchi automaton build from ψ using Definition 5. The language of $\mathcal{A}_\psi$ is the set of models of ψ.*

The interested reader is referred to Pelánek and Strejcek (2005) for more details. We conclude this section with an example.

Example 11. Consider the LTL formula $\psi = p\,\mathsf{U}\,((\mathsf{G}\,q) \wedge (\mathsf{F}\,\neg r))$ over $\mathsf{AP} = \{p, q, r\}$. Obviously, the formula is in negation normal form, and the sub-formulas are p, q, r, $\neg r$, $\mathsf{G}\,q$, $\mathsf{F}\,\neg r$, $((\mathsf{G}\,q) \wedge (\mathsf{F}\,\neg r))$, and ψ itself. Figure 3.5 shows the alternating automaton that results from applying the construction of Definition 5 to this formula. Note that only three states (and the virtual "**true**" state) are reachable from the initial state, and another five states are not reachable and can thus be stripped from the automaton without changing the language. ★

Note that the alternating automaton has a special tree-shaped structure, which ensures that along every branch in the run tree, occurrences of the same state along a run/branch in the run tree have to be in direct succession. We also call such automata *very-weak* or *one-weak*.

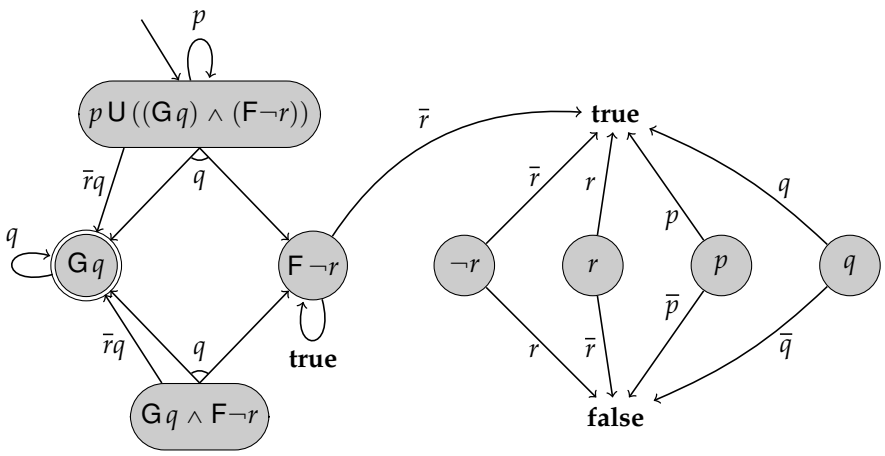

Figure 3.5: The alternating automaton from Example 11. Note that the Boolean functions of the transition function have been simplified prior to drawing them. In particular, conjunctions with **true** and disjunctions with **false** have been collapsed.

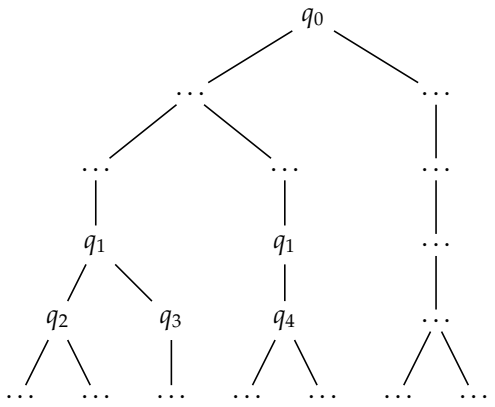

Figure 3.6: Example run tree of an alternating automaton, used for explanatory purposes in Section 3.3.

3.3 Translating an alternating Büchi automaton to a non-deterministic Büchi automaton

To complete translation workflow from LTL to non-deterministic Büchi automata, it remains to be shown how we can translate an alternating Büchi automaton to a non-deterministic Büchi automaton (with the same language). The prodecure for doing so (Miyano and Hayashi, 1984) is commonly known under the name *Miyano-Hayashi construction*. We first explain the idea behind the construction, then state the construction itself and conclude with an example.

Consider the run tree of an alternating automata that is depicted in Figure 3.6. On the fourth level of the tree, we have two nodes that are labelled by some state q_1. We can see immediately that the despite this same labelling, the two subtrees that are rooted at these two nodes are not the same. We call such run trees *non-uniform*. Formally, we call a run tree $\langle T_r, \tau_r \rangle$ *uniform* if for every two nodes t_1 and t_2, we have that if $|t_1| = |t_2|$ and $\tau(t_1) = \tau(t_2)$, then the sub-trees induced by t_1 and t_2 are the same.

While a non-uniform run tree such as the one from Figure 3.6 might be valid for the alternating automaton under concern, the question is natural whether we actually need to consider such run trees when checking whether there exists an accepting run tree for a given word. Indeed, as for such a run tree to be accepting, both the sub-trees must respect the transition function of the automaton and be accepting, we could replace one of these sub-trees by the other one and would still obtain a valid run tree that is accepting if the old one is accepting too. We can repeat this operation an arbitrary number of times while keeping the run tree valid and preserving acceptance. However, this does not automatically

mean that we only need to consider uniform run trees, as a for a non-uniform run tree to be made uniform, an infinite number of such sub-tree substitutions might need to be performed, and we do not know if the validity and acceptance of the tree is preserved under this many operations. Luckily, Miyano and Hayashi (1984) were however able to show this. The interested reader is referred to their work for more information.

The Miyano-Hayashi construction makes use of this uniformity argument. The idea is to encode accepting run trees of the alternating Büchi automaton that we want to translate into runs of the non-deterministic Büchi automaton. The state space of the latter keeps track of in which states we can be along a run tree branch for the alternating automaton. For every transition, we use non-determinism in the target automaton to guess which transitions are taken along the branches in the run tree of the alternating automaton. For a given word, we can thus read a valid run tree of the alternating automaton off the states of the non-deterministic Büchi automaton that occur along an accepting run of the latter. Additionally, to have our non-deterministic Büchi automaton accept only words that have an accepting run tree in the alternating automaton, we check that there is no infinite branch in the run tree on which accepting states occur only finitely often. We do this using a *break-point construction* that keeps track of along which branches in the run tree we have not "recently" seen accepting states. Once there is no such branch left, we reset our "recently" set, at which point we have reached a so-called *break-point*). We construct our non-deterministic automaton such that the states that witness break-points are the accepting ones. If and only if we visit break-points infinitely often, the run of the non-deterministic automaton is accepting and all branches in the corresponding run tree of the alternating automaton are accepting. Let us now formalise this idea.

Definition 6. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be an alternating Büchi automaton. We define its translation to a non-deterministic Büchi automaton by $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, \mathcal{F}')$ with:*

- $Q' = 2^Q \times 2^Q$
- $Q'_0 = \{(\{q_0\}, \varnothing)\}$
- $\mathcal{F} = \{(S, S') \mid S' = \varnothing\}$
- $\delta((S, S'), x) = \{(S'', S''') \mid \forall q \in Q : (q \in S) \rightarrow (S'' \models \delta(q, x)) \wedge (q \in S') \rightarrow ((S''' \cup (S'' \cap \mathcal{F})) \models \delta(q, x)), S'' \supseteq S'''\}$ *for all $S, S' \subseteq Q$ with $S' \neq \varnothing$ and $x \in \Sigma$*
- $\delta((S, \varnothing), x) = \{(S'', S''') \mid \forall q \in Q, (q \in S) \rightarrow (S'' \models \delta(q, x)), S''' = S'' \backslash \mathcal{F}\}$ *for all $S \subseteq Q$ and $x \in \Sigma$*

Theorem 2 (Miyano and Hayashi, 1984)**.** *Let $\mathcal{A}$ be an alternating Büchi automaton, and $\mathcal{A}'$ be the non-deterministic Büchi automaton built from $\mathcal{A}$ by the construction of Definition 6. The languages of $\mathcal{A}$ and $\mathcal{A}'$ are the same.*

We conclude with an example that explains the construction.

Example 12. Consider the alternating Büchi automaton $\mathcal{A}$ in Figure 3.7 and $w = \varnothing\varnothing\{a\}\varnothing\varnothing\{b\}\{a\}\ldots$ as some word over its alphabet. Figure 3.8 contains an accepting run tree for this run, along with the run of the translated non-deterministic Büchi automaton $\mathcal{A}'$ obtained by applying the Miyaho-Hayashi construction. From what we can tell, the run tree of the alternating automaton could be accepting, and so could the run of the non-deterministic automaton. On the other hand, consider the run tree and non-deterministic automaton run for the same word in Figure 3.9. Here, we take a transition from q_1 to q_2 too early, and thus end up with a branch of the run tree in which q_3 is never left, which necessarily leads to a non-accepting run tree. For the corresponding run in $\mathcal{A}$, we can see that then, the second element in the states' tuples along the run never becomes empty and accepting states can then not be visited any more.

There is no figure that represents $\mathcal{A}'$ itself - it would be too large to be of use.

Note that in the Miyano-Hayashi construction, we do not track along which branch of the alternating automaton we have reached a state, as this is simply not necessary. ★

3.4 Summary and discussion of complexities

In this chapter, we have discussed a complete flow to translate LTL formulas to non-deterministic Büchi automata, as needed for model checking, and useful for synthesis. The reader desiring a broader

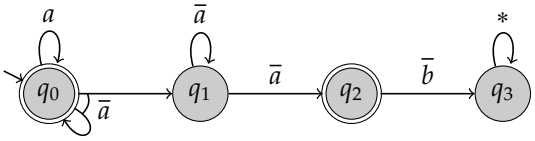

Figure 3.7: Example for an alternating Büchi automaton that is equivalent to the LTL property $\mathsf{GF}(a \vee \mathsf{XX}b)$. Conjunctive branching is depicted by an arc. In order to make the run trees in Figure 3.8 and Figure 3.9 interesting, the automaton is a bit more complicated than needed.

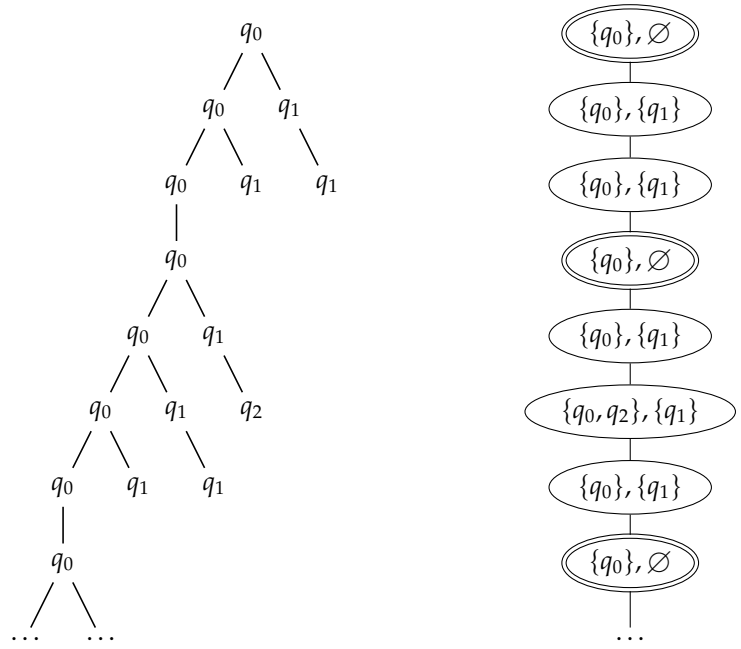

Figure 3.8: A run tree of the alternating automaton from Figure 3.7 for $w = \varnothing\varnothing\{a\}\varnothing\varnothing\{b\}\{a\}\ldots$ as the input word, and the corresponding run in the non-deterministic Büchi automaton build from the automaton in Figure 3.7 using the Miyano-Hayashi construction

introduction into the world of automata over infinite words might also want to have a look at a tutorial by Vardi (1995).

Let us conclude by summarising the complexities and blow-ups involved in the process. Translating the LTL formula into negation normal form may lead to a blow-up of a factor of two, at most. Then, we translate the formula to an alternating automaton of the same size as the formula. Finally, we have an exponential blow-up of the automaton size when translating it to non-deterministic acceptance mode. All in all, we obtain an exponential blow-up. Since all steps are simple, the time and space complexities of the conversion process are also exponential. It can be shown that an exponential blow-up from LTL to non-deterministic Büchi automata is unavoidable, but for brevity, we will omit this proof here.

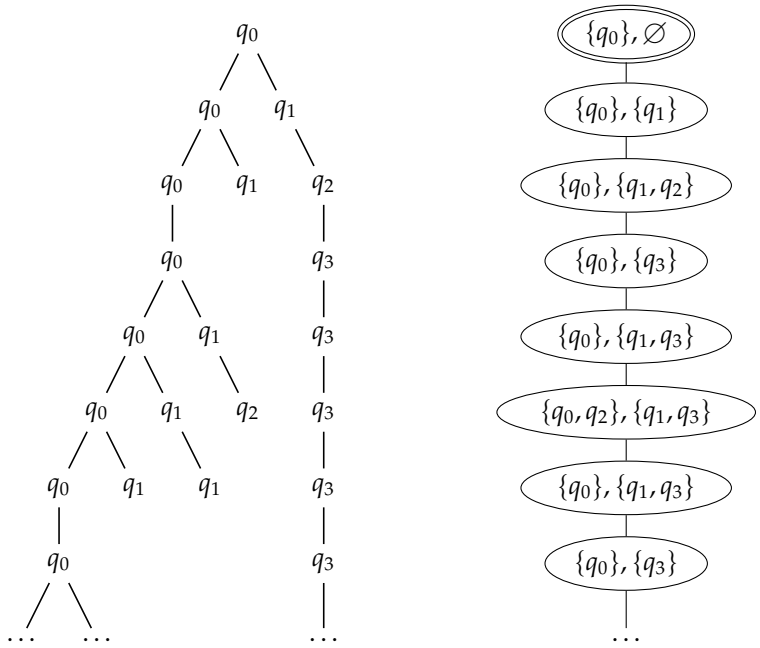

Figure 3.9: A (non-accepting) run tree of the alternating automaton from Figure 3.7 for $w = \varnothing\varnothing\{a\}\varnothing\varnothing\{b\}\{a\}\ldots$ as the input word, and the corresponding run in the non-deterministic Büchi automaton build from the automaton in Figure 3.7 using the Miyano-Hayashi construction

Chapter 4

Model checking and synthesis with Büchi automata

In this chapter, we apply the concepts and techniques from the previous chapters to perform model checking and synthesis. We start with the former, where the concepts from the previous chapters will prove to be sufficient. For the latter, we will obtain a solution that is sound, but not complete. An analysis of the shortcomings of the theory so far for synthesis will guide the way to the additional concepts needed for this purpose that we discuss in the next chapter.

The discussions in this chapter allow us to connect the concepts needed for synthesis with those for model checking, and thus not only help those readers that are already familiar with model checking to find a path into the synthesis problem, but also provide insight into why the concepts commonly used for synthesis are defined in the way they are.

4.1 Model checking

Let us take a look at a classical construction for model checking, where we test that all traces induced by a Mealy or Moore machine are contained in some word language whose negation is represented by a non-deterministic Büchi automaton.

Without loss of generality, we will be dealing with Mealy machines as systems here. We start by giving a construction for model checking these, and then discuss how the non-determinism of the specification automaton is tackled in the construction.

Definition 7. *Let $\mathcal{M} = (S, \Sigma^I, \Sigma^O, \delta_{\mathcal{M}}, s_0)$ be a Mealy machine and $\mathcal{A} = (Q, \Sigma, \delta_{\mathcal{A}}, Q_0, \mathcal{F})$ be a non-deterministic Büchi automaton with $\Sigma = \Sigma^I \times \Sigma^O$.*

We define the product automaton *of $\mathcal{M}$ and $\mathcal{A}$ as the non-deterministic Büchi automaton $\mathcal{A}' = (Q', \Sigma^I, \delta', Q_0', \mathcal{F}')$ with:*

- $Q' = Q \times S$
- $Q_0' = Q_0 \times \{s_0\}$
- $\mathcal{F}' = \mathcal{F} \times S$
- *For all $(q, s) \in Q \times S$ and $x \in \Sigma^I$, we have $\delta'((q, s), x) = (\delta_{\mathcal{A}}(q, (x, y)), s')$ for $\delta_{\mathcal{M}}(s, x) = (s', y)$.*

The idea of the product automaton is captured in the following theorem:

Theorem 3. *Given a Mealy automaton $\mathcal{M}$, a Büchi automaton $\mathcal{A}$ and the product automaton $\mathcal{A}'$ taken from these two, we have that $\mathcal{A}'$ accepts precisely the (input) words that induce a trace of $\mathcal{M}$ that is accepted by $\mathcal{A}$.*

Proof. Note that the state space of $\mathcal{A}'$ is the product of the state space of the machine $\mathcal{M}$ and the state space of the Büchi automaton $\mathcal{A}$. The construction ensures that for every character in a word, first the Mealy machine is updated by taking the character as the input and the corresponding output is obtained, and then, the Büchi automaton state is updated by taking a transition for this input/output combination. The non-determinism of the Büchi automaton is also present in the product automaton. At the same time, the definition of the accepting states ensures that the runs corresponding to words accepted by $\mathcal{A}$ have corresponding runs in $\mathcal{A}'$. □

This theorem can be used for model checking a reactive system. Given a Mealy machine and a Büchi automaton that accepts the traces that are *not* allowed by our specification, we can build the product of

these and obtain a Büchi automaton that accepts the input sequences for which our machine behaves in an undesired manner. Thus, by checking if the product automaton accepts *any* word, we can test if the system satisfies the specification. If we do not find such a word, then we are fine. If we do find such a word, then we also obtain an input word in which the system behaves in a faulty way. Starting from an LTL formula ψ, we can obtain such an automaton by translating $\neg\psi$ using the construction of the previous chapter. What remains to be done is to describe how we can check a Büchi automaton for emptiness.

For this, recall that accepting runs are the ones on which an accepting state is visited infinitely often. If we find an accepting run, then we also find an accepted word by reading it from the edge labels. For a run to be accepting, it must visit at least *one* accepting state infinitely often. This means that there has to exist a path in the automaton from the state back to itself, i.e., a sequence of states $q_1, q_2, \ldots, q_n$ such that $q_n = q_1$, $q_1 \in F$, and for every $i \in \{1, \ldots, n-1\}$, there exists some $x \in \Sigma$ with $q_{i+1} \in \delta(q_i, x)$. Furthermore, q_1 must be reachable somehow from an initial state of the automaton. The cycle $q_1, q_2, \ldots, q_n$ and the path from the initial state to q_1 in combination are also called an *accepting lasso* in the automaton. Once we fixed an $q_1 \in \mathcal{F}$, finding a path from q_1 back to q_1 can be performed in time linear in the sizes of Q and Σ. Thus, by iterating over the states in $\mathcal{F}$, and checking the existence of a lasso cycle from the state and a path from the initial state to q_1, we can probe the automaton for emptiness. In fact, testing an automaton for emptiness can even be done a bit more efficiently, i.e., in time linear in the alphabet size and linear in the number of states of the automaton (see Baier and Katoen, 2008 for details).

It is interesting at this point to discuss the following question: how is non-determinism handled in this construction? We have started with a Mealy automata with non-deterministic input and a Büchi automaton with non-deterministic transitions, and created a Büchi automaton with non-deterministic transitions, which we need to check for language emptiness. For the latter, we search for a lasso in the automaton, consisting of a cycle on which an accepting state is visited at least once, and a path to this cycle. The search for this lasso allows us to resolve the non-determinism in a simple finitary manner, and the lasso both represents the word that the automaton needs to read to produce the accepting run, and the run itself.

Now what about the case in which we have a Büchi automaton that does not represent the negation of the specification (i.e., the *bad traces* that we do not want our system to have), but rather the specification itself (i.e., an automaton that accepts the traces that our system to check is allowed to have)? It turns out that there is no construction with the same low complexity as above (which was: linear in the size of the alphabet, linear in the number of states of the specification automaton, and linear in the number of states of the Mealy machine). Indeed, even for a one-state Mealy machine and a two-letter alphabet, it can be shown that the model checking problem is PSPACE-complete in the number of states of the automaton representing the specification. So this problem is significantly harder, although it looks quite similar. The reason why it is harder is rooted in the fact that the *types of non-determinism* we have in the problem are different.

Reconsider the simple model checking procedure from above: we check if there *exists* a word and a corresponding run that is accepting in the product automaton. On the other hand, if we are given a Büchi automaton for the positive specification, then we check if there *exists* a word such that *all* runs of the specification automaton for the word are non-accepting. Only the latter problem has an alternation between universal and existential quantification, and this is what causes the high complexity.

As a summary, what we achieved in the construction above by negating the specification is to make the *types of non-determinism* compatible with each other.

4.2 Synthesis – a first encounter

We have just seen that Büchi automata are a suitable specification model to verify finite-state systems against a specification. In the model checking construction, a product between a finite-state system and a Büchi automaton is built and checking for emptiness is performed by testing if there is some accepting lasso in the automaton. The emptiness test could in principle also be used for synthesis, but this time we do not search for bad words, but for good ones: we start with a Büchi automaton for the (non-negated) specification, and then check if regardless of the input to the system, we can build an accepting lasso. Since a reactive system cannot look into the future, we have to take into consideration that an output in the lasso must only depend on the input observed so far. This idea leads naturally to the notion of games.

We start by discussing the idea for *safety specifications* that intuitively say that "something bad should never happen", and give examples and intuitions for this simplified case. After becoming comfortable with games, we are ready to dive into the more complicated winning conditions, such as the Büchi one, which is needed for the idea above in order to accommodate all Büchi automata. As we will see however, the resulting synthesis construction is sound, but not complete. We will analyse why this is the case, and how the problem can be avoided. This will provide us with an intuitive understanding of the ideas of the complete synthesis constructions in the next chapters.

4.2.1 Safety games

Games are a commonly used model in computer science to represent the interaction between a system and its environment. As such, many synthesis approaches utilise them either implicitly or explicitly. We are interested in synthesis from ω-regular specifications, so we consider games with ω-regular winning conditions. In these games, two players make their turns for an indefinite amount of time and move a *pebble* over a game structure of finite size. Whether the first or the second player wins the game depends on the positions visited infinitely often along the play.

The connection to synthesis is made by assigning one player the function of simulating the environment that feeds input to a system, while the other player takes the role of the system, and produces output. The two players play an indefinite amount of time, modelling the indefinite computation time of a reactive system. By ensuring that the game structure and the winning condition lets the *system player* win if and only if the *play*, i.e., the sequence of game positions that is the outcome of the two players' decisions, represents a trace that is in the specification of the system, we ensure that a *strategy* for the system player to win the game represents a system that satisfies the specification.

In games with an ω-regular winning condition, it can be shown that one of the players always has a strategy to win the game, i.e., to enforce a play that is winning for the respective player. If the game is winning for the system player, then the strategy represents an implementation of the system that is always correct, regardless of the input to the system. This is also the connection to *tree automata*, which we will get to know in Chapter 5: as the environment player can play *any* sequence of moves, the strategy of a system player is tree-shaped, with the branching corresponding to the input. Furthermore, the system player must win for every input sequence.

Let us start the discussion by formally defining *safety games*.

Definition 8. *A* safety game *is defined as a tuple* $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{\text{in}})$ *with the sets of game positions (also called vertices)* V^0 *and* V^1*, the alphabets (or action sets)* Σ^0 *and* Σ^1*, the transition functions* $E^0 : V^0 \times \Sigma^0 \to (V^1 \cup \{\bot\})$ *and* $E^1 : V^1 \times \Sigma^1 \to (V^0 \cup \{\bot\})$*, and the initial position* $v_{\text{in}} \in V^0$*. We say that* V^0*,* Σ^0 *and* E^0 *belong to player* 0*, while* V^1*,* Σ^1 *and* E^1 *belong to player* 1*.*

Given a decision sequence $\rho^0 = \rho^0_0\rho^0_1\rho^0_2\ldots$ *for player* 0 *and a decision sequence* $\rho^1 = \rho^1_0\rho^1_1\rho^1_2\ldots$*, we say that* ρ^0 *and* ρ^1 *induce a play* $\pi = \pi^0_0\pi^0_1\pi^1_0\pi^1_1\ldots$ *in* $\mathcal{G}$ *such that* $\pi^0_0 = v_{\text{in}}$ *and for every* $i \in \mathbb{N}$ *and* $p \in \{0,1\}$*, we either have* $\pi^p_i = \bot$ *and* $\pi^{1-p}_{i+p} = \bot$*, or* $\pi^{1-p}_{i+p} = E_p(\pi^p_i, \rho^p_i)$*.*

During the course of a play, whenever a player makes a next decision (and thus gradually constructs the decision sequence observed), it can take the past decision of both players into account. We formalise this idea by defining strategies*. We call a function* $f^0 : (\Sigma^0 \times \Sigma^1)^* \to \Sigma^0$ *a strategy for player* 0*, and a function* $f^1 : (\Sigma^0 \times \Sigma^1)^* \times \Sigma^0 \to \Sigma^1$ *a strategy for player* 1*. A pair of decision sequences* (ρ^0, ρ^1) *is said to be in correspondence to* f^0 *if for every* $i \in \mathbb{N}$*,* $\rho^0_i = f^0(\rho^0_0\rho^1_0\rho^0_1\ldots\rho^1_{i-1})$*. Likewise, a pair of decision sequences* (ρ^0, ρ^1) *is said to be in correspondence to* f^1 *if for every* $i \in \mathbb{N}$*,* $\rho^1_i = f^1(\rho^0_0\rho^1_0\rho^0_1\ldots\rho^0_i)$*. We say that a play* π *corresponds to a strategy* f^0/f^1 *if there exists a pair of decision sequences* (ρ^0, ρ^1) *that induces* π *and for which* ρ^0 *is in correspondence to* f^0/ρ^1 *is in correspondence to* f^1*, respectively.*

For a safety game, we declare one of the players to be the safety player*, and the other player is the* reachability player*. The safety player wins a play if along the play,* $\bot$ *is never visited. If* $\bot$ *is ever visited along the play, the reachability player wins instead.*

If for a strategy f^0*, all plays corresponding to* f^0 *are winning for player* 0*, we say that* f^0 *is a winning strategy for player* 0*. If player* 0 *has a winning strategy, we say that* $\mathcal{G}$ *is winning as player* 0 *can enforce by choosing her moves according to* f^0 *that all plays that might occur when playing the strategy* f^0 *are winning for player* 0*. The corresponding definitions for player* 1 *are analogous.*

Note that the fact that we assumed that the initial position belongs to player 0 does not influence the generality of our game model: we can always introduce an additional vertex v'_{in} into V^1, and set

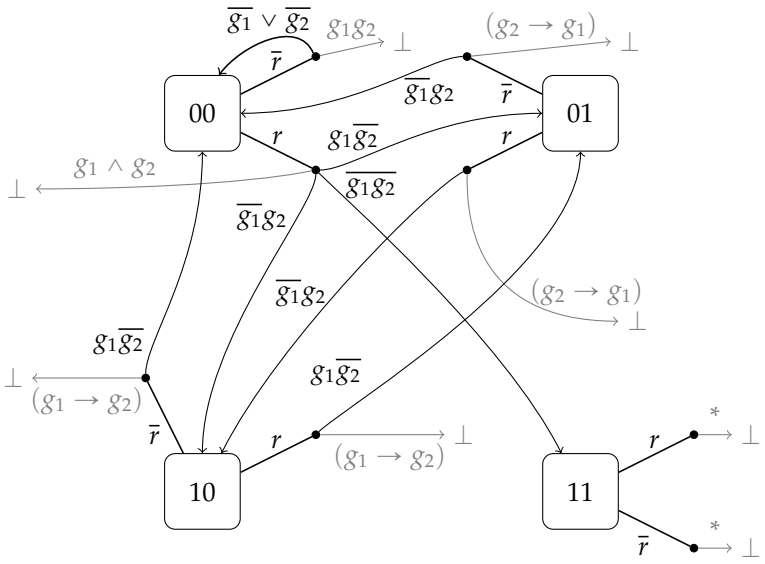

Figure 4.1: An example safety game. Vertices of player 0 are depicted by rectangles, whereas vertices of player 1 are little filled circles. Edges are labelled by the alphabet symbol they refer to. We use Boolean constraints to characterise sets of alphabet symbols.

Round		0	1	2	3	4	5	6	7
Play		00 (00, ∅)	00 (00, {r})	10 (10, ∅)	00 (00, {r})	01 (01, ∅)	00 (00, {r})	11 (11, {r_1})	⊥
Σ^0	r	0	1	0	1	0	1	0	…
Σ^1	g_1	0	0	1	1	0	0	0	…
	g_2	1	1	0	0	1	0	1	…

Table 4.1: Example decision sequences and the corresponding play for the game from Example 13.

$E^0(v_{\text{in}}, x) = v'_{\text{in}}$ for any $x \in \Sigma^0$. This way, we have effectively restricted player 0 to choose a decision that leads to v'_{in} in every possible play from its initial position, and thus we forwarded the first decision that actually matters to player 1.

After all these formal definitions, let us discuss a little example to clarify the connection of games to the synthesis problem.

Example 13. Consider the game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{\text{in}})$ depicted in Figure 4.1 with $V^0 = \{00, 01, 10, 11\}$, $\Sigma^0 = 2^{\mathsf{AP}^I}$ for $\mathsf{AP}^I = \{r\}$, $\Sigma^1 = 2^{\mathsf{AP}^O}$ for $\mathsf{AP}^O = \{g_1, g_2\}$, and $v_{\text{in}} = 00$. For this game, let us assume that player 1 is the safety player (i.e., player 1 wants to avoid visiting $\bot$ during the course of the play), and that the positions of player 1 are simply the combinations of the V^0 predecessor and the last chosen action of player 0. This way, for example, we have $E^0(00, \varnothing) = (00, \varnothing)$.

Every play in the game starts in position 00, and player 0 is the first to choose some action (from Σ^0). Let us assume that player 0 chooses the action $\varnothing$. The decision sequence for player 0 thus starts with $\rho_0^0 = \varnothing$. Now, player 1 makes some move, like for example $\{g_1\}$. Then, we can fix already the first three elements of any play in the game for which $\rho_0^0 = \varnothing$ and $\rho_0^1 = \{g_1\}$ are the first actions of the two players in the game: $\pi = (00)\,(00, \varnothing)\,(00)$. The remaining play is built in the same way as in the first round and player 0 and 1 always alternate in making their moves.

Table 4.1 depicts a pair of possible continuation decision sequences of the two players for this example and the resulting play. ★

Let us read the decision sequences of the two players in a word-like manner. The sequences from Table 4.1 would then be read as the word $w = \{g_2\}\{r, g_2\}\{g_1\}\{r, g_1\}\ldots$. Using this word view onto the decision sequences, we can see that the game in Example 13 is built in a special way: precisely the words on which the LTL specification

$$\begin{aligned} \psi \quad &= \quad \mathsf{G}(\neg g_1 \vee \neg g_2) \\ &\wedge \quad \mathsf{G}(r \to (g_1 \vee \mathsf{X} g_1)) \end{aligned}$$

$$\wedge \quad \mathsf{G}(r \rightarrow (g_2 \vee \mathsf{X} g_2))$$

does not hold correspond to plays that eventually visit $\bot$. Thus, a winning strategy for the safety player 1 in this game, that can make its moves depending only on the previous moves of player 0, can be seen as a system implementation that satisfies the specification. As a consequence, it is reasonable to call $\mathcal{G}$ the *synthesis game* of ψ for the interface $(\{r\}, \{g_1, g_2\})$.

Before we discuss how to solve a safety game, let us shortly formalise the idea of making a safety game that corresponds to a specification, i.e., that ensures that (1) all winning plays represent words that satisfy the specification, and (2) if and only if a specification is *realisable* (i.e., there exists an implementation with the given interface for the given specification), we find a winning strategy for the system player (here: the safety player). Starting from a deterministic automaton, we call this operation *spreading* the automaton.

Definition 9. *A deterministic safety automaton is a Büchi automaton with only one initial state for which for every state/character combination, there is at most one successor, and for which all states are accepting. In a deterministic safety automaton tuple, we can thus leave out the set of accepting states.*

Given a deterministic safety automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ and some alphabets Σ_I and Σ_O such that $\Sigma = \Sigma_I \times \Sigma_O$, we call a game $\mathcal{G} = (V^0, V^1, \Sigma_I, \Sigma_O, E^0, E^1, v_{in})$ the spreading *of $\mathcal{A}$ over Σ^I/Σ^O if the following conditions hold:*

- $V^0 = Q \cup \{\bot\}$, $V^1 = Q \times \Sigma^I$
- *For all $v \in V^0$, $x \in \Sigma^I$, we have $E^0(v, x) = (v, x)$*
- *For all $(v, x) \in V^1$, $y \in \Sigma^O$, we have $E^1((v, x), y) = v'$ for $\delta(v, (x, y)) = \{v'\}$ if $\delta(v, (x, y)) \neq \varnothing$, and $E^1((v, x), y) = \bot$ otherwise*
- $v_{in} = q_0$

In this definition, we split the transitions in the deterministic automaton into edges in the game for the input and the output of the system, and introduce suitable intermediate positions. This ensures that plays in the game essentially represent runs in the original automaton. A play leads to position $\bot$ if and only if any infinite word that starts with the decision sequences for the play up to reaching position $\bot$ is rejected by the automaton. We call such words *bad prefixes* for the language represented by the automaton. At the same time, the game represents correctly that the system can generate some carefully chosen output, but must work for any input, and the input is fed to the system step-by-step. Thus, a game that represents the spreading of a deterministic safety automaton specification is indeed suited for reducing the synthesis problem for a specification to game solving.

4.2.2 Solving safety games

Let us now have a look at the question how to *solve* a safety game, i.e., how to determine which of the players has a strategy to win the game. For doing so, we have to establish that the question which player wins the game is actually well-posed, i.e., there actually is a player that has a winning strategy. Classes of games that always have this property are called *determined*. It has been proven by Martin (1975, 1982) for a very large class of games, with winning condition types that go far beyond safety (and finite position sets), that determinacy is assured. However, for simple winning condition types, like safety (and the *parity winning condition* that we will encounter in Chapter 6.1), determinacy even follows from the proofs of correctness for corresponding game solving algorithms, as it will here.

Consider a safety game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ and a prefix play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \dots \pi_n^p$ such that player p is the reachability player and there exists a move $x \in \Sigma^p$ such that $E_p(\pi_n^p, x) = \bot$. In this case, by playing x, the reachability player can ensure that the play is winning for her. In fact, this applies to all plays that eventually visit position π_n^p. So the only way to have a winning strategy for the safety player is to avoid having a play visit π_n^p altogether. We call such game positions *bad positions* (for the safety player).

Now consider the converse case: assume that we have a prefix play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \dots \pi_n^p$ such that player p is the safety player, and for all $x \in \Sigma^p$, we have $E_p(\pi_n^p, x) = \bot$. In this case, we can also call π_n^p a bad position, as the safety player cannot avoid visiting $\bot$ in the postfix play.

The bad positions identified by the two cases above are also called the reachability player's *one-step attractor* of $\bot$, as whenever a play is a state that is identified to be bad by the cases above, the reachability

player can enforce to visit $\bot$ after at most one step in the game. For simplicity, we include $\bot$ in the one-step attractor of $\bot$.

There is also the case that a position can be considered to be bad as from it, the reachability player can ensure to win within n steps in the game for $n > 1$. For $n = 2$, these are reachability player positions that have a successor that lies in the one-step attractor of $\bot$ and safety player positions that only have successors in the one-step attractor of $\bot$. It can be shown by induction that these are the only cases that we need to consider when computing the *two-step attractor* of $\bot$.

The argument can be continued for arbitrary values of n. If for a reachability player position, there exists a successor in the $(n-1)$-step attractor of $\bot$, or for a safety player position, all successors are in the $(n-1)$-step attractor of $\bot$, then a position is in the n-step attractor of $\bot$. We defined the n-step attractor of $\bot$ to always include the $(n-1)$-step attractor of $\bot$ by definition, so for increasing values of n, the n-attractor of $\bot$ can only become larger. For games with a finite number of positions, as we discuss here, the attractor at some point *saturates*, i.e., for some value m, the m-step and $(m+i)$-step attractor of $\bot$ coincide for every $i \in \mathbb{N}$.

It turns out that after the attractor saturates, for the remaining positions (the *good* positions), if we start a play in any of these, then the safety player has a strategy to win, and in particular any strategy is winning that avoids bad positions (which are the positions in the m-step attractor of $\bot$). Let us formalise the notions of attractors before proving this fact.

Definition 10. *Let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ be a safety game. Given a set of positions $P \subseteq V^0 \uplus V^1$ and a player $p \in \{0,1\}$, we define the one-step p-attractor of P as:*

$$\begin{aligned} \mathrm{Attr}_p^1(P) &= \{v \in V^p \mid \exists x \in \Sigma^p : E^p(v,x) \in P\} \\ &\cup \{v \in V^{1-p} \mid \forall x \in \Sigma^{1-p} : E^p(v,x) \in P\} \\ &\cup P \end{aligned}$$

For $i > 1$, we define the i-step attractor recursively as:

$$\mathrm{Attr}_p^i(P) = \mathrm{Attr}_p^{i-1}(\mathrm{Attr}_p^1(P))$$

As for every $p \in \{0,1\}$, $\mathrm{Attr}_p^1 : 2^{(V^0 \uplus V^1)} \to 2^{(V^0 \uplus V^1)}$ is a function such that $\mathrm{Attr}_p^1(P) \subseteq P$ for every $P \subseteq (V^0 \uplus V^1)$, and $(V^0 \uplus V^1)$ is finite for all games considered, for any set P, repeatedly applying Attr_p^1 to this set eventually leads to a stable set P', i.e., for some $m \in \mathbb{N}$, we have $P' = \mathrm{Attr}_p^m(P) = \mathrm{Attr}_p^{m+i}(P)$ for every $i \in \mathbb{N}$.

We define Attr_p^∞ to be the function that maps every $P \subseteq (V^0 \uplus V^1)$ to $\mathrm{Attr}_p^m(P)$ for some $m \in \mathbb{N}$ such that $\mathrm{Attr}_p^m(P) = \mathrm{Attr}_p^{m+i}(P)$ for every $i \in \mathbb{N}$.

Theorem 4. *Let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ be a game and $B = \mathrm{Attr}_p^\infty(\{\bot\})$ be the multi-step attractor set of the reachability player p. The reachability player wins the game, i.e., can ensure that a play eventually visits $\bot$ if and only if $v_{in} \in B$.*

Proof. It is not difficult to prove by induction that for every $i \in \mathbb{N}$, $\mathrm{Attr}_p^i(\{\bot\})$ contains precisely the set of game positions from which, when starting there, player p can ensure to eventually visit $\{\bot\}$. Since $\mathrm{Attr}_p^\infty(\{\bot\}) = \mathrm{Attr}_p^m(\{\bot\})$ for some $m \in \mathbb{N}$, for any $v_{in} \in B$, the game is winning for the reachability player after at most m steps.

Now assume that $v_{in} \notin B$. The construction ensures that $((V^0 \uplus V^1) \backslash B)$ is a player $(1-p)$-*paradise*, i.e., for every player-p position in $(V^0 \uplus V^1) \backslash B)$, all successor positions are in $((V^0 \uplus V^1) \backslash B)$, and for every player $(1-p)$-position in $((V^0 \uplus V^1) \backslash B)$, there exists a successor position in $((V^0 \uplus V^1) \backslash B)$. Thus, from any position in $((V^0 \uplus V^1) \backslash B)$, player $1-p$ can enforce that the next position is again in $((V^0 \uplus V^1) \backslash B)$. Since $((V^0 \uplus V^1) \backslash B)$ does not contain $\bot$, this means that by player $1-p$ simply enforcing that the next game position is in $((V^0 \uplus V^1) \backslash B)$ whenever player $1-p$ can make a move, it is ensured that the play never visits $\bot$ and thus is winning for player $1-p$.

Since we have a partitioning of the game positions into the ones from which player 0 can win when starting there and the ones from which player 1 wins, we have shown determinacy of safety games along the way. □

The theorem gives rise to an algorithm to determine the set of positions in a game from which the reachability player wins: start with $\{\bot\}$ as the set of bad positions and iteratively extend the set of bad

positions by adding the one-step attractor set (for the reachability player) of the previously found bad positions until a fixed point is reached. The resulting position set is the full set of bad positions for the safety player.

We say that the reachability player wins *from* or *in* any of the bad positions, as the player can ensure that every play that visits a bad position at some point will also eventually visit $\bot$. For the remaining positions, the safety player wins from or in them.

Example 14. Let us determine the set of bad positions of the game in Figure 4.1. The vertices of the safety player, which is player 1 here, are given as small dots, whereas the vertices of the reachability player (player 0) are depicted as rectangles.

The one step-attractor of $\bot$ first of all contains $\bot$ itself, but as $\bot$ is also the only successor of position 11, that position is contained in the one-step attractor of $\bot$ as well. Further applications of the attractor function do not find more positions to be bad, and the remaining positions are therefore found to be winning for the safety player. ★

In the literature, the expression *solving a game* often refers to partitioning the positions in the game into the ones from which player 0 has a strategy to win if the game starts there, and the ones from which player 1 has a strategy to win. Since for doing so, there is no need to define an initial vertex in the game, it is often omitted from the game tuples. The remaining elements together are also called a *game structure*. Our (simple) algorithm effectively solves a game.

It remains to discuss how we can actually obtain a winning strategy (and a finite representation of it) for any of the players. Intuitively, if v_{in} is winning for the reachability player p, then we have $v_{in} \in B$ for the set of bad positions B with $v_{in} \in \mathrm{Attr}^p_k(\{\bot\})$ for some $k \in \mathbb{N}$. Let us define w.l.o.g. k to be the minimal value in $\mathbb{N}$ such that $v_{in} \in \mathrm{Attr}^p_k(\{\bot\})$. If player p ensures that for every $0 \leqslant i \leqslant k$, the ith element in a play is in $\mathrm{Attr}^p_{k-i}(\{\bot\})$ (when defining $\mathrm{Attr}^p_0(\{\bot\}) = \{\bot\}$), then the play visits $\bot$ after at most k decisions of the two players. Note that player $1-p$ cannot avoid advancing in the attractor sequence – we have defined the attractor precisely in a way such that a game position of player $1-p$ (the safety player) is only contained in $\mathrm{Attr}^p_{k-i}(\{\bot\})$ if all successors of the position are in $\mathrm{Attr}^p_{k-i-1}(\{\bot\})$. Likewise, the construction ensures that along the play, player p always has the possibility to move forward along the attractor chain. Thus, any strategy that ensures moving along the attractor chain is a winning one for the reachability player.

For the safety player, the situation is simpler. As we have seen in the proof of Theorem 4, any strategy that ensures to not leave the set of *good positions* (the complement of the bad positions) is winning.

For both reachability and safety player in a safety game, whenever one of these players has a winning strategy, then the player also has a *positional strategy*, i.e., the strategy only depends on which position the play is in at a point in time. For a winning safety strategy, this means that for every of her positions in the game, whenever a play reaches the position, the player chooses the same next action. For a winning reachability strategy, this means that a position is never visited twice – building a strategy by the construction described above ensures this. Positional strategies are of special interest in the theory of games with an ω-regular winning condition as they are naturally representable in a finitary manner, namely by a map from positions to actions to be taken. Let us now formalise this notion.

Definition 11. *Let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ be a game. For every $p \in \{0,1\}$, we call a function $f^p : V^p \to \Sigma^p$ a* memoryless strategy *for player p.*

A memoryless strategy f^0 for player 0 induces a general strategy f'^0 of player 0 by defining $f'^0(t) = f(E^1(E^0(\dots E^1(E^0(v_{in}, t^0_0), t^1_0) \dots, t^0_n), t^1_n))$ for every $t = t^0_0 t^1_0 t^0_1 \dots t^1_n \in (\Sigma^0 \times \Sigma^1)^*$. Likewise, a memoryless strategy f^1 for player 1 induces a general strategy f'^1 of player 1 by defining $f'^1(t) = f(E^0(E^1(\dots E^1(E^0(v_{in}, t^0_0), t^1_0) \dots, t^1_{n-1}), t^0_n))$ for every $t = t^0_0 t^1_0 t^0_1 \dots t^1_{n-1} t^0_n \in (\Sigma^0 \times \Sigma^1)^* \times \Sigma^0$.

By abuse of notation, we do not distinguish between the two strategy representations for game types that always permit memoryless strategies in the following.

Using the definition of positional strategies, we can now represent a winning strategy for the safety player p in a game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ in a finitary way. For this, let B be the set of bad vertices in $V^0 \uplus V^1$. By setting $f^p(v)$ to some arbitrary $x \in \Sigma^p$ such that $E^p(v, x) \notin B$ for every $v \in V^p$ whenever such a value x exists, and to some arbitrary other value otherwise, we obtain a winning strategy for player p in $\mathcal{G}$ (if $v_{in} \notin B$). Computing a winning strategy for the reachability player can be performed similarly, with the difference that progress towards the $\bot$ state has to be ensured in every step.

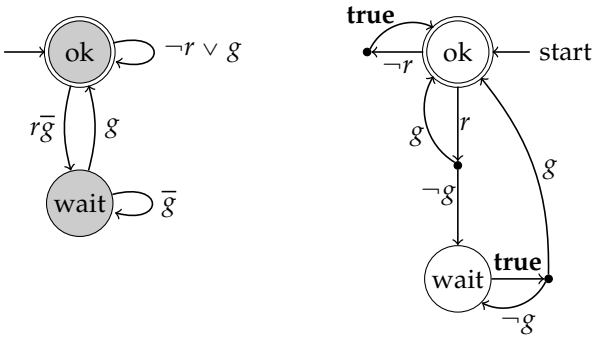

Figure 4.2: An example Büchi automaton and its spreaded Büchi game

4.2.3 Büchi games

We have seen in Example 13 how games and the synthesis problem are connected: if we can build a game such that precisely the decision sequences that are winning for the system player are the ones that, when read as words, satisfy the specification, then the result of solving the game tells us if the specification under concern has an implementation that satisfies it or not. We have introduced a *spreading* step in Definition 9 that allows us to build such games from specification automata.

Safety games however do not allow us to directly encode *liveness properties* into the game. Intuitively, a liveness property states that "something good must happen infinitely often". Assume for example that in the specification of an *arbiter*, which has the task to mediate access to a shared resource between a set of clients, we have the liveness property that *grants should be given infinitely often* by the arbiter. We can neither encode this requirement as a safety property, nor as a reachability property (which then the non-safety player in a safety game could try to fulfil). Thus, it is not clear how to encode such a requirement into a safety game. Classically, this problem is solved by stepping away from the simple safety winning condition, and upgrading to a more complex winning condition.

We introduce *Büchi games* by means of an example. Consider the automaton depicted on the left-hand side of Figure 4.2. If we spread it in the same way as we did for safety games, and assume $\Sigma^0 = 2^{\{r\}}$ and $\Sigma^1 = 2^{\{g\}}$, we obtain the game on the right-hand side of the figure. The graphical notation is the same as for safety games, with the addition that this time, some positions are accepting. We take those positions to be accepting for which the respective states in the automaton are accepting. Plays that visit accepting positions infinitely often are then winning for the Büchi player in the game (which is the system player for the scope of this chapter). It can be observed that precisely the set of plays that are winning for the system player are the ones that satisfy the LTL property $\psi = \mathsf{G}(r \rightarrow \mathsf{F}g)$ when read as words. Interpreting r as a "request" signal, and g as a "grant" signal, what we get is a specification for a one-client arbiter: whenever we get a request, then eventually later, we need to get a grant. Clearly, this specification is winning for the system player: a system implementing ψ could always give grants, regardless of whether requests have been issued. This is reflected in the game by the fact that the system player can always choose $\{g\}$ as her move such that the accepting position "ok" is never left. Thus, a play induced by such a strategy visits accepting positions infinitely often and is winning for the Büchi player.

In the example above, we have started with a deterministic Büchi automaton, and spread it to a game. Recall that we have shown that non-deterministic Büchi automata are sufficient to describe any LTL property. So if we had some way of spreading a non-deterministic Büchi automaton to a Büchi game as well, the only thing left to discuss would be how to solve Büchi games, and we would be done with the synthesis problem for the scope of this introduction.

We will discuss the Büchi game solving problem in the next subsection. Unfortunately, spreading a non-deterministic Büchi automaton in a way that is both sound and complete is not as simple as spreading a deterministic Büchi automaton. In fact, we will see later in Chapter 6.2 that from a complexity-theoretic point of view, it cannot be done in a way as we spread deterministic automata. We discuss in this section *why* simply applying the spreading idea from deterministic automata to non-deterministic ones does not work. This insight is tremendously helpful for discussing the sound and

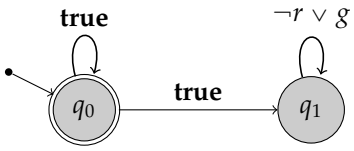

Figure 4.3: A non-deterministic Büchi automaton for the LTL property $\mathsf{FG}(r \rightarrow g)$

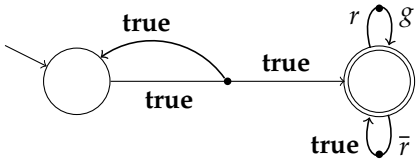

Figure 4.4: A simple (almost-)Büchi game.

complete synthesis approach in the next chapter, as it shows us on which aspect to focus in order to solve the synthesis problem in its full generality.

If we want to spread a non-deterministic automaton into a game, we will need to let one of the two players resolve the non-determinism, i.e., pick which transition to take for input/output combinations for which there is more than one. As the non-system player has an interest in letting the other player lose the game, and thus has an incentive to pick the wrong transitions, the only sane choice here is to leave this task to the system player. Note that strictly speaking, the games that we get in this way do not conform to the definition of games we have, namely that for every position/action combination in the edge relation of the system player, there is precisely one successor position, as now, we can have more than one successor. We can however ignore this problem for the time being, as actual game solving algorithms can easily cope with this modification.

Let us discuss the idea of spreading a non-deterministic automaton with an example. Consider an LTL specification $\psi = \mathsf{FG}(r \rightarrow g)$. The specification states that at some point along a run, from that point onwards, g is always given whenever r is given. Figure 4.3 shows a Büchi automaton for this specification. Let us now spread this automaton over the alphabet $2^{\{r,g\}}$ by taking the input alphabet $2^{\{r\}}$ and the output alphabet $2^{\{g\}}$, and assigning the choice of which transition to take whenever non-determinism needs to be resolved to the system player. The result can be seen in Figure 4.4.

We can see from this game that in fact not all plays in this game that, when reading their corresponding decision sequences as words, satisfy ψ, are winning for the Büchi player. For every word, including the ones satisfying ψ, there is a corresponding play that stays in the left position, and is thus losing for the Büchi player in the game. Thus, in such a game, we require the system player not only to ensure that the decision sequences produced by the two players, when read as a word, satisfy the specification, but also to resolve the non-determinism in the transitions to take in a way that witnesses this. If the system/Büchi player has a strategy in a game to win, the game however ensures that the decision sequences corresponding to the strategy satisfy the specification. Thus, as long as for the system player, the game is winning, we can solve the synthesis problem in this way. It can be formally proven that this method of letting the system player resolve the non-determinism is sound, i.e., ensures that whenever for a synthesis game built in this way, there is a winning strategy for the system player, then the winning strategy represents an implementation that satisfies the specification from which we built the game.

Unfortunately, this method is not complete. The problem is that the system player cannot always resolve the non-determinism in an eager way without losing the game. Again, let us discuss this phenomenon by means of an example. Figure 4.5 shows a non-deterministic Büchi automaton for the specification $(\mathsf{GF}r \wedge \mathsf{GF}g) \vee (\mathsf{FG}\neg r \wedge \mathsf{FG}\neg g)$. It can be proven that there does not exist a deterministic Büchi automaton for the same language, so for building a game, we would start with a non-deterministic one. Assume now that we have $\Sigma^0 = 2^{\{r\}}$ and $\Sigma^1 = 2^{\{g\}}$, i.e., the first (non-Büchi) player controls the input to the system, which is, whether r is set or not, whereas the second (Büchi) player controls g. For this input/output signature, the specification is realisable: the output signal can simply copy the value of r to g in every computation cycle in order to satisfy the specification. Let us now spread the automaton to a game (depicted in Figure 4.6). The game is **not** winning for the Büchi player, as the Büchi player

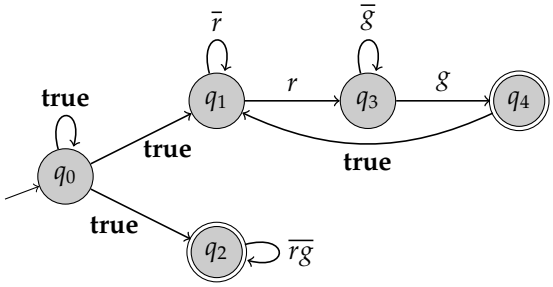

Figure 4.5: A non-deterministic Büchi automaton build for the specification $(\mathsf{GF}r \wedge \mathsf{GF}g) \vee (\mathsf{FG}\neg r \wedge \mathsf{FG}\neg g)$.

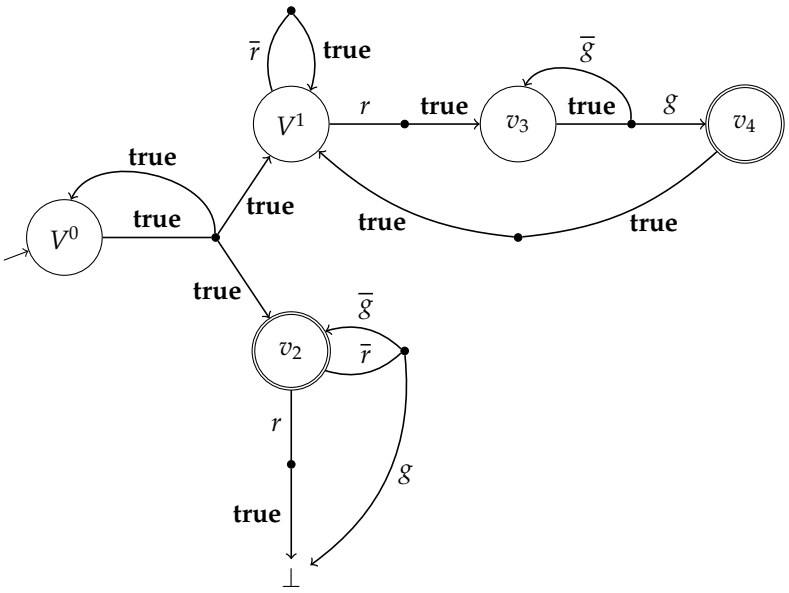

Figure 4.6: A Büchi game depicting the straight-forward spread of the automaton from Figure 4.5

has to move away from the initial position at some point in order to visit accepting positions infinitely often, but would have to look into the future for guessing correctly whether to move to q_1 or q_2. When moving to q_1, if the input is $\varnothing$ for the rest of the play, then the play eventually gets stuck in q_1 and is losing for the Büchi player. If the Büchi player chooses to move to q_2, then getting $\{r\}$ as the next input lets her lose the game. So there is no winning strategy for the Büchi player even though there exists an implementation for the specification.

The reason why the Büchi player loses this game is that she has to guess in advance which state of the Büchi automaton can be a state that is visited infinitely often along a run of the automaton for the decision sequences in the game. However, as the input comes in step-by-step, and the Büchi player is not clairvoyant in the game, she cannot do this. This example shows that the different types of non-determinism we may have in synthesis (non-deterministic input and non-deterministic branching with respect to the automaton transition followed) do not play well with each other. Thus, in order to perform synthesis in a complete, and not only sound, manner, we will need to ensure by the approach that we choose for synthesis that the types of non-determinism are compatible with each other. Since we cannot remove the non-determinism in the input, as it is an integral part of every reactive system, we will need to step away from the idea of spreading a non-deterministic automaton to a game.

4.2.4 Solving Büchi games

Before moving on to setting the scene for a complete synthesis approach, let us quickly discuss how to solve Büchi games for applications in which they are sufficient (e.g., for synthesis when the specification is given as a deterministic Büchi automaton).

First of all, observe that if the Büchi player has a winning strategy, then from every position that can be reached along the strategy, she must be able to enforce that accepting positions are visited infinitely often in the following. It thus must never leave the set of positions X from which this can be enforced. This gives rise for the following idea: first compute the set of positions X' from which the Büchi player can enforce to visit an accepting position at least once in the future. Then, we take this set of positions and compute the set of positions from which she can enforce visiting an accepting position at least twice (i.e., at least once while being in X' afterwards). Both of these position sets are obviously supersets of X. We repeat the idea for an increasing value of n. The sets computed can only become smaller over time, but reach a fixed point at some point. Observe that this is our state set X: By induction, if the position set from which we can enforce visiting an accepting position n times is the same as for $n + 1$ times, it will also be the same as for $n + 2$ and $n + 3$ times, and so on. Thus, we can enforce visits to accepting states arbitrarily often, and since we only have a finite number of positions, also infinitely often. With the knowledge established previously in this section, we can easily perform all of these computations.

For obtaining a winning strategy, recall how we can obtain a winning strategy for the reachability player in a safety game: we follow the attractor to the set of bad states step-wise. We can apply the same main idea here: From all positions in X, we can build an attractor to visit the next accepting position, without leaving X. If the Büchi player always follows this attractor, she is guaranteed to visit accepting states infinitely often if the game is winning for her. This fact also immediately leads to the observation that positional strategies suffice for the Büchi player in a Büchi game. Actually, positional strategies also suffice for the other player if she is winning, but we do not prove that here.

Chapter 5

Tree languages, tree automata and the connection between model checking and synthesis

On the first look, the model checking construction from the previous chapter appears to be a bit weird from a conceptual point of view: it only works if we *negate* the specification beforehand, and then, it takes a Mealy automaton that represents the whole computation tree of a reactive system, and quantifies over all of its branches to test them against some linear-time specification. The synthesis construction that we had, on the other hand, used the non-negated specification. Thus, the question is natural whether there is no conceptually simpler way of describing the model checking problem that works with a non-negated specification. We will see in this chapter that the answer is positive, and indeed the conceptual simplification allows us to connect the model checking and synthesis problems in a clear way.

The alternative model checking construction consists of two parts: first of all, we remove the necessity to negate the specification by introducing a new word automaton type that allows us to represent the non-negated specification in a form that is amenable to model checking. Then, we move from the world of word languages to *tree languages* to get rid of the explicit quantification of tree branches.

In fact, tree languages are the connecting point of model checking and synthesis of reactive systems. While model checking amounts to checking if a given computation tree is contained in a tree language, synthesis amounts to testing if a given tree language is empty, and if it is not, to obtain a computation tree in the language. The tree language represents the specification in both cases. We will introduce *tree automata* as finitary models for tree languages. The tree languages will replace the games introduced for synthesis in the previous chapter for the time being, and we will later reconnect them again.

Note that for clarity, henceforth, we will always say whether an automaton we are concerned with is a tree or word automaton. All automata that we have seen in the previous chapters have been of word type.

5.1 Universal co-Büchi word automata

In Section 3.1, we have seen in the context of alternating automata that universal branching for word automata is an interesting concept: in this branching mode, we can split up a run of the automaton and require that for both of the resulting parts, the run is accepting. In model checking, we also quantify universally over traces of the system and check that the trace conforms to the specification. By using alternating automata with only universal branching as automaton model for representing the specification for model checking, we can thus make the *types of quantification* compatible. In order to be still able to specify all properties that are representable as non-deterministic Büchi automata, we however have to complement the acceptance condition of our alternating automata to *co-Büchi* type. Let us formalise this idea.

Definition 12. *A* universal co-Büchi word automaton *(*UCW*) is given as a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ with the set of states Q, the alphabet Σ, the transition function $\delta : Q \times \Sigma \to 2^Q$, the initial state q_0, and the set of* rejecting states *$\mathcal{F}$.*

Given a word $w = w_0 w_1 w_2 \in \Sigma^\omega$, we say that some tree $\langle T_r, \tau_r \rangle$ is a run tree of $\mathcal{A}$ for w if $T_r \subseteq \mathbb{N}^*$, $\epsilon \in T_r$, $\tau_r : T_r \to Q$, $\tau_r(\epsilon) = q_0$, and for all $t \in T_r$ with $\tau_r(t) = q$, we have that $\{\tau(tx) \mid x \in \mathbb{N}, tx \in T_r\} = \delta(q, w_{|t|})$. Note that for every word, there exists a unique run tree (up to node renaming and unnecessary sub-tree duplication).

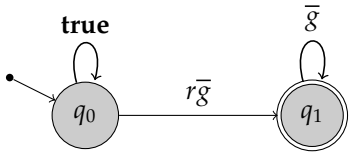

Figure 5.1: A universal co-Büchi word automaton that is equivalent to the LTL formula $\mathsf{G}(r \rightarrow \mathsf{F}g)$. Rejecting states are doubly-circled.

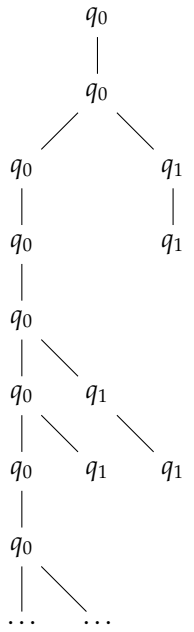

Figure 5.2: The run tree of the universal co-Büchi word automaton from Figure 5.1 for the word $w = \varnothing\,\{r\}\,\varnothing\,\{r,g\}\,(\{r\}\,\{r\}\,\{g\})^\omega$.

We say that w is accepted by $\mathcal{A}$ if for the unique run tree $\langle T_r, \tau_r \rangle$ for $\mathcal{A}$ and w, along every branch, there are only finitely many occurrences of labels containing rejecting states of $\mathcal{A}$.

As customary in the literature on automata theory and synthesis, from this chapter onwards, we use three-letter abbreviations such as "UCW" to denote automaton types. The first letter always stands for the branching mode, i.e., "U" for universal branching, "N" for non-deterministic branching, and "D" if we have a deterministic automaton. The second letter denotes the acceptance condition, where "B" represents a Büchi acceptance condition, whereas "C" denotes a co-Büchi acceptance condition. The last letter tells us if a word automaton ("W") or a tree automaton ("T"), which we define in the next section, is meant.

Example 15. Consider the universal co-Büchi automaton shown in Figure 5.1. The automaton has the set of models of the LTL formula $\mathsf{G}(r \rightarrow \mathsf{F}g)$ for $\mathsf{AP} = \{r, g\}$ as its language. Figure 5.2 represents a run tree for the automaton and the word $w = \varnothing\,\{r\}\,\varnothing\,\{r,g\}\,(\{r\}\,\{r\}\,\{g\})^\omega$. ★

Note that the UCW's graphical depiction in Example 15 is the same as of a non-deterministic Büchi word automaton for the complement language. This is no coincidence – as a UCW only accepts words for which every branch in the run tree (that contains all possible runs of the automaton for the input word) visits the rejecting states only finitely often, and a non-deterministic Büchi word automaton accepts the words for which there exists a run that visits accepting states infinitely often, they are dual cases.

This also tells us immediately how to obtain a UCW for some LTL formula: negate the formula, compute the non-deterministic Büchi word automaton for it and interpret the automaton tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$ obtained as a UCW. Recall from Chapter 3.1 that without loss of generality, we can easily switch between

automaton definitions with only one initial state and automaton definitions with many initial states, as the conversion is simple and may only lead to at most one additional state.

Note that the fact that we have to consider run trees for UCWs instead of just runs as for non-deterministic Büchi automata suggests that model checking with the former is technically more complicated than with the latter. However, this is actually not true: the run trees do not do more than listing all the possible runs. Furthermore, when we move to tree languages to connect model checking and synthesis and construct tree automata for representing tree languages, we will need run trees anyway.

5.2 Tree languages and tree automata

Recall that we want to reformulate the model checking problem as as a problem of tree containment in a tree language, as opposed to building the product of a Mealy machine and a non-deterministic Büchi automaton as in Chapter 4.1. This requires us to roll out the machine to a computation tree, and to translate the word language, represented by a word automaton (here, a UCW), to a tree language. We have seen in Chapter 2.3 how to do the former, so we only need to discuss the latter here.

In both model checking and synthesis, it is common that we want to use linear-time properties as specifications. To obtain tree language from those, we need to perform a *spreading* step first.[1]

Given a word language L over some alphabet Σ and some sets Σ^I and Σ^O with $\Sigma = \Sigma^I \times \Sigma^O$, we call the tree language $L' = \{\langle (\Sigma^I)^*, \tau\rangle \mid \forall w_0^I w_1^I w_2^I \ldots \in (\Sigma^I)^\omega.(w_0^I, \tau(w_0^I))(w_1^I, \tau(w_0^I w_1^I))(w_2^I, \tau(w_0^I w_1^I w_2^I)) \ldots \in L$ the *Mealy-type spreading* of L over the interface (Σ^I, Σ^O).

Likewise, given a word language L over some alphabet Σ and some sets Σ^I and Σ^O with $\Sigma = \Sigma^I \times \Sigma^O$, we call the tree language $L' = \{\langle (\Sigma^I)^*, \tau\rangle \mid \forall w_0^I w_1^I w_2^I \in (\Sigma^I)^\omega.(w_0^I, \tau(\epsilon))(w_1^I, \tau(w_0^I))(w_2^I, \tau(w_0^I w_1^I))$ the *Moore-type spreading* of L over the interface (Σ^I, Σ^O).

In these definitions, we have essentially taken the definitions of traces from computation trees in the Mealy- and Moore-type semantics, and required the computation trees in the tree language to only have traces in the word language.

The spreading description above was so far only on a conceptual level. If we want to spread the language of an automaton over words and want to obtain a finitary description of the resulting tree language, we will need to introduce a suitable formalism for the latter, which is precisely what *tree automata* do.

Tree automata come in a variety of flavours. We will take the UCWs from the previous section and spread them to *universal co-Büchi tree automata* (*UCT*s). Spreading non-deterministic Büchi automata is on the other hand not so easy and omitted here. We will see later *why* this is not easy (Chapter 6.2). We start with a formal definition of UCTs and a formal description of the run-trees of a UCT. Then, an example will clarify the definitions.

Definition 13. *A universal co-Büchi tree automaton (UCT) is defined as a tuple $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0, \mathcal{F})$ with the set of states Q, the* branching alphabet Σ^I, *the* label alphabet Σ^O, *the transition function $\delta : Q \times \Sigma^O \to 2^{Q \times \Sigma^I}$ and the set of rejecting states $\mathcal{F}$.*

A UCT $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0, \mathcal{F})$ accepts or rejects full Σ^O-labelled Σ^I-trees, i.e., trees of the form $\langle (\Sigma^I)^*, \tau\rangle$ with $\tau : (\Sigma^I)^* \to \Sigma^O$. For doing so, it computes a run tree $\langle T_r, \tau_r\rangle$ from the computation tree under concern. Formally, a run tree is defined as follows:

- $T_r \subset \mathbb{N}^*$
- $\tau_r : T_r \to Q \times (\Sigma^I)^*$
- $\tau_r(\epsilon) = (q_0, \epsilon)$
- *For all $t_r \in \mathbb{N}^*$ with $\tau_r(t_r) = (q, t)$, we have $\delta(q, \tau(t)) = \{(q, x) \mid \exists n \in \mathbb{N} : t_r n \in T_r, \tau(t_r n) = (q, tx)\}$.*

We call a run tree $\langle T_r, \tau_r\rangle$ accepting if for all infinite branches $t = t_0 t_1 t_2 \ldots \in \mathbb{N}^*$ in $\langle T_r, \tau_r\rangle$, we have $\inf(\tau_r(\epsilon)|_Q\ \tau_r(t_0)|_Q\ \tau_r(t_0 t_1)|_Q\ \tau_r(t_0 t_1 t_2)|_Q \ldots) \cap \mathcal{F} = \varnothing$.

[1] Note that the term *spreading* for this step is uncommon in the synthesis literature. Most publications just skip this step or treat it implicitly. Kupferman et al. (2006c) call this step computing the tree automaton for the tree language *derived* from the word language. Since the word *derived* is used for many other things as well, we however use the term *spreading* here as it is unambiguous.

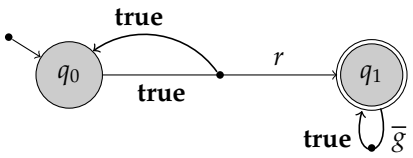

Figure 5.3: Universal co-Büchi tree automaton (UCT) for Example 16. The intermediate dots on the edges represent the distinction between directions in the tree (the inputs) and tree node labels (the outputs). When reading a node label $x \in \Sigma^O$, one needs to consider all transitions whose label up to the dot match with x. So, the edge label **true** from state q_0 to the small intermediate node right of it actually represents two outputs: g and $\neg g$. The doubly-circled states are the rejecting states of the UCT.

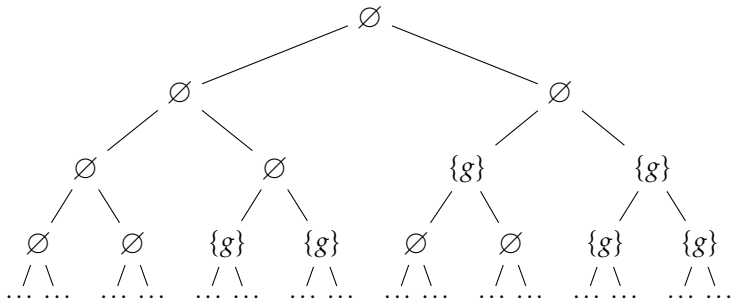

Figure 5.4: Example computation tree over the direction set $\Sigma^I = 2^{\{r\}}$ and the label set $\Sigma^O = 2^{\{g\}}$, used in Example 16.

Example 16. Let us discuss an example of a universal co-Büchi tree automata. Figure 5.3 describes a simple example UCT $\mathcal{A}$ over $\Sigma^O = 2^{\{g\}}$ and $\Sigma^I = 2^{\{r\}}$. For the same direction/node label set, there is a computation tree $\langle T, \tau \rangle$ given in Figure 5.4. We can compute the unique run tree $\langle T_r, \tau_r \rangle$ of $\mathcal{A}$ and $\langle T, \tau \rangle$, which is described in Figure 5.5. Note that for UCTs, unlike for alternating word automata, run trees are always unique (up to node renaming and unnecessary sub-tree duplication).

Formally, the transition function of the UCT $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_{in}, \mathcal{F})$ has $\delta(q_0, x) = \{(\varnothing, q_0), (\{r\}, q_0), (\{r\}, q_1)\}$ for $x \in \{\varnothing, \{g\}\}$. Thus, along the run tree for the UCT, we always stay in q_0 in one branch for every input sequence. At the same time, q_1 is only reachable along the $\{r\}$ direction.

Intuitively, if we interpret r as a *request* signal, and g as a *grant* signal, the UCT represents the specification that every request should eventually be answered by a grant afterwards. This can be seen from the fact that q_1 is the only rejecting state, q_1 is only entered after r is read as input from the initial state, and a branch stays in q_1 until it eventually sees a g as output. So the UCT checks that along every branch in a computation tree, the LTL specification $\mathsf{G}(r \rightarrow \mathsf{F}g)$ holds in the Mealy-type computation model, or equivalently, $\mathsf{G}(r \rightarrow \mathsf{XF}g)$ holds in the Moore-type computation model. ★

Now we have defined the necessary automaton types automata for both word and tree languages in model checking and synthesis, it remains to be explained how we can spread a UCW to a UCT. As we defined the spreading operation for languages in a different way for Mealy- and Moore-type computation models, our construction also needs to take this into account.

When spreading a word automaton, we must ensure that the new automaton accepts precisely the trees that only have branches that represent traces in the language of the word automaton. Universal branching allows us to *simulate* the runs of the original word automaton along every branch.

Definition 14. *Given a universal word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ and some interface (Σ^I, Σ^O) with $\Sigma = \Sigma^I \times \Sigma^O$, we define the* spread *of $\mathcal{A}$ over (Σ^I, Σ^O) in the* Moore-type *semantics as the universal tree automaton $\mathcal{A} = (Q', \Sigma^I, \Sigma^O, \delta', q_0', \mathcal{F}')$ with:*

- $Q' = Q$
- *For all $q \in Q'$, $y \in \Sigma^O$, $\delta'(q, y) = \{(q', x) \in Q' \times \Sigma^I \mid q' \in \delta(q, (x, y))\}$*

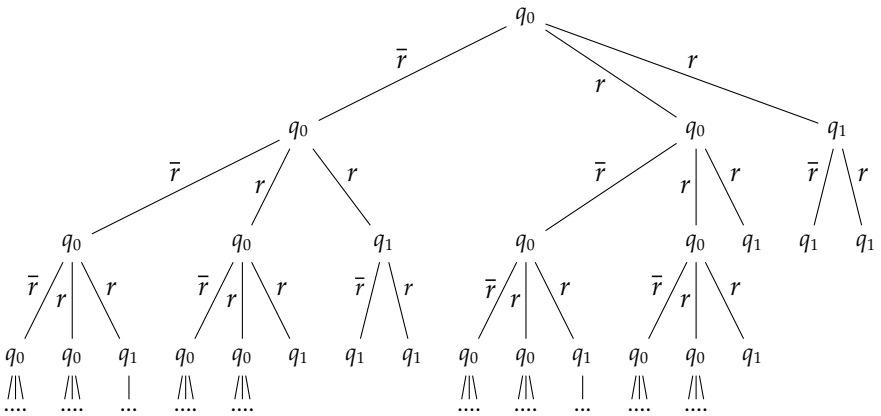

Figure 5.5: A run tree of the UCT in Figure 5.3 for the computation in Figure 5.4. Formally, run tree nodes of a UCT are labelled by a UCT state and a computation tree node to which the run tree node refers to. We have distributed the latter onto the edge labels. For example, a run tree node that is reached by the edges $\bar{r}$ and then r has $\varnothing\{r\}$ as the second element of its node labelling.

- $q_0' = q_0$
- $\mathcal{F}' = \mathcal{F}$

Definition 15. *Given a universal word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ and some interface (Σ^I, Σ^O) with $\Sigma = \Sigma^I \times \Sigma^O$, we define the* spread *of $\mathcal{A}$ over (Σ^I, Σ^O) in the* Mealy-type *semantics as the universal tree automaton $\mathcal{A} = (Q', \Sigma^I, \Sigma^O, \delta', q_0', \mathcal{F}')$ with:*

- $Q' = Q \times \Sigma^I \uplus \{q_0'\}$
- *For all $(q, x) \in Q'$, $y \in \Sigma^O$, $\delta'((q, x), y) = \{((q', x'), x') \mid q' \in Q, x' \in \Sigma^I, q' \in \delta(q, (x, y))\}$*
- *For all $y \in \Sigma^O$, $\delta(q_0', y) = \{((q_0, x), x) \mid x \in \Sigma^I\}$*
- $\mathcal{F}' = \mathcal{F} \times \Sigma^I$

From a conceptual point of view, the spreading operation appears to be more complicated for the Mealy-type semantics, as here, we need $|\Sigma^I|$ times as many states. In practice, however, this does not matter: any model checking or synthesis algorithm can compute the spreading on-the-fly and depending on the concrete operations performed, both problems can easily be equally hard in both types of semantics.

5.3 Model checking with tree automata

Let us now discuss the model checking problem for universal co-Büchi tree automata and finite-state systems. Recall that the latter can be characterised by the set of *regular* computation trees, i.e., computation trees that only have a finite number of distinct sub-trees. For word properties representable in linear-time temporal logic, we have seen that they can be described by universal co-Büchi word automata, and by spreading the automaton, we obtain a universal co-Büchi tree automaton (UCT) that accepts the computation trees that only have branches on which the specification is satisfied.

To actually model check a regular computation tree against a UCT, we will need to check if there exists a branch in the run tree of the UCT for the computation tree on which a rejecting state is visited infinitely often. The fact that UCTs have a unique run tree for every input computation tree makes this particularly simple. Let t_r and t_r' be nodes in a run tree $\langle T_r, \tau_r \rangle$ for the computation tree $\langle T, \tau \rangle$. If $\tau_r(t_r) = (q, t)$ and $\tau_r(t_r') = (q, t')$ for some state q of the UCT and some $t, t' \in T$ such that t and t' induce the same sub-tree in $\langle T, \tau \rangle$, then the sub-trees of $\langle T_r, \tau_r \rangle$ that are induced by t_r and t_r' also have to be the same. Since we only have finitely many states in the UCT and finitely many sub-trees in $\langle T, \tau \rangle$, along

a branch in a run tree that tells us the that computation tree is rejected, for some state q in the UCT and some $t \in T$, there have to be infinitely many nodes $\{t_r^i\}_{i\in\mathbb{N}}$ such that $\tau_r(t_r^i) = (t', q)$ for some t' that induces the same sub-tree in $\langle T, \tau\rangle$ as t, and rejecting states in the UCT are visited infinitely often along the branch. In fact, we can assume that in between two tree nodes t_r^i and t_r^{i+1}, we visit a rejecting state, as the non-existence of such a family $\{t_r^i\}_{i\in\mathbb{N}}$ would imply that that the tree is accepted (which can be proven by assuming the converse and deriving a contradiction to the fact that $\langle T, \tau\rangle$ is regular). Note that in general, there will be several different families $\{t_r^i\}_{i\in\mathbb{N}}$ that satisfy these constraints.

Using the facts above, we can derive a decision procedure for a regular computation tree $\langle T, \tau\rangle$ and a UCT to check if the former is contained in the language of the latter: build the run tree for this setting, but let every branch end after it contains a pair of nodes t_r and t_r' with $\tau_r(t_r) = (q, t)$ and $\tau_r(t_r') = (q, t')$ for some state q of the UCT and some $t, t' \in T$ such that t and t' induce the same sub-tree. If between a pair of such nodes, some rejecting state is visited, then we know that there exists a branch in the full run tree that results from repeating the part between t_r and t_r' infinitely often and that witnesses non-acceptance of the computation tree. If on the other hand, we do not find such a pair, then there does not exist a branch in the full computation tree that has a minimal family $\{t_r^i\}_{i\in\mathbb{N}}$ in the sense above. Thus, there cannot be a non-accepting branch in the run tree for $\langle T, \tau\rangle$ and thus, $\langle T, \tau\rangle$ is accepted by the UCT.

Our prefix computation tree can have at most the size $|Q|^2$ times the number of equivalence classes of $\langle T, \tau\rangle$ to the square, where Q is the state set of the UCT. Thus, if we have an efficient way of computing the equivalence classes of $\langle T, \tau\rangle$, computing successors from some equivalence class, and testing the tree for the existence of a branch on which an accepting state is visited in between two visits to the same state/equivalence class combination, we obtain a polynomial-time model checking method. When unrolling a Mealy- or Moore-machine on-the-fly, we can use the state set of the machine as equivalence class set, and then essentially apply the algorithm for searching for accepting lassos from Chapter 4.1 again, where the first occurrence of some state/equivalence class combination represents the start of the lasso cycle. This then reduces the complexity of the procedure to being linear.

Overall, we obtain a model checking complexity that is linear in the size (number of states) of the UCT, the number of states of the Mealy- or Moore machine, and in the sizes of the input and output alphabets, which is the same as in Chapter 4.1. However, in the presentation in this chapter, we abstracted from whether we are in the Mealy- or Moore-type computation model, and deferred the necessary distinction to the unrolling definition of the computation tree of the system.

5.4 Synthesis

After we have connected the model checking problem to checking inclusion of a (regular) computation tree in the language of a UCT, we now turn towards synthesis, which amounts to checking *emptiness* of the language of a UCT. If the language of the UCT is not empty, then we also want to find a computation tree that is in the automaton's language.

It can be shown that the complexity of checking a UCT language for emptiness is considerably higher than testing for a given regular computation tree (represented as a finite-state machine) if it is contained in the language of the UCT. While we have seen in the previous section that the latter problem is solvable in polynomial time, UCT emptiness checking is EXPTIME-hard. This can be shown by reducing the acceptance of an alternating polynomially-space-bounded (APSPACE) Turing machine onto the UCT emptiness problem. By the fact that APSPACE=EXPTIME (Chandra et al., 1981), this fact follows.

The most common way to solve the emptiness problem of tree automata languages is to employ *determinism* in the automaton structure: if for every state q in the UCT $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0, \mathcal{F})$ and every $i \in \Sigma^I$ and $o \in \Sigma^O$, there is at most one $q' \in Q$ such that $(q', i) \in \delta(q, o)$, then we call $\mathcal{A}$ deterministic. Note that without loss of generality, for deterministic automata, we can assume that there is precisely *one* such q' by introducing a non-rejecting additional absorbing *sink state*.

The nice property of deterministic tree automata is that we can build *games with ω-regular winning conditions* that serve as computation model for checking the emptiness of the tree automaton from them, as we will see in the next chapter.

Unfortunately, not all UCT can be made deterministic. One way to solve this problem is to move to richer modes of acceptance than the co-Büchi acceptance mode, like for example *parity acceptance*, which we will get to know in Chapter 6.1. As an initial synthesis approach to be presented, we will however apply a different idea that has been introduced under the name *bounded synthesis* (Schewe and Finkbeiner, 2007) in the literature. It is technically simpler than approaches based on more complex

acceptance modes, and thus allows us to give a full description of it in the scope of this introduction. Furthermore, many recent efficient synthesis tools build upon the bounded synthesis idea. Thus, the approach is also a very modern one.

Let us take for granted that for *every* ω-regular word property of an alphabet Σ and every interface (Σ^I, Σ^O) with $\Sigma = \Sigma^I \times \Sigma^O$, whenever there exists a reactive system with that interface that satisfies the property, then there also exists a finite-state one. This statement implies that we can restrict our quest for checking if there exists a reactive system satisfying the specification to checking if there exists a suitable finite-state reactive system without losing precision. Note that the statement is actually true – we will however not prove it in this introduction. Technically, the fact follows from the concepts that we will learn in Chapter 6.1, but there are many ways to show this. For example, Vardi (1995) gives a self-contained derivation of this fact by employing a richer automaton class, named *Rabin automata*.

For a finite state machine $\mathcal{M}$ with n states that is accepted by some UCT $\mathcal{A}$ with m states, we know that along every branch in the (extended) run tree for $\mathcal{M}$ and $\mathcal{A}$, there are at most $n \cdot m \cdot |\Sigma^I|$ visits to rejecting states along the branch, as otherwise by looping the directions between two nodes in the run tree with an extended node labelling that occurs twice along the branch (when replacing the run tree node in the extended labelling by its equivalence class), and where in between the two nodes, a rejecting state is visited, we obtain a branch on which rejecting states are visited infinitely often.

It can be shown that for a UCT with n states, if the UCT has a non-empty language, there exists a finite-state machine with at most $2^{O(n \log n)}$ states whose computation tree is accepted by the UCT. Combining this fact with the idea from the previous paragraph, we obtain that it suffices to search for computation trees for which every branch in the corresponding run tree in the UCT visits rejecting states at most $n \cdot |\Sigma^I| \cdot 2^{O(n \log n)}$ times. This fact follows from Corollary 1, which we will discuss later (on page 69). The insight from this fact is that we can build a *deterministic safety tree automaton* for checking this *bound* on the number of visits to rejecting states.

Definition 16. *A deterministic safety tree (DST) automaton is defined as a tuple $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0)$ with the set of states Q, the input/branching alphabet Σ^I, the output/labelling alphabet Σ^O, the transition function $\delta : Q \times \Sigma^O \to 2^{(Q \cup \{\bot\}) \times \Sigma^I}$, and the initial state $q_0 \in Q$. They are similar to universal co-Büchi tree automata, with the extension that $\bot$ is a special state that a transition can lead to, and the restriction that the tree automaton needs to be deterministic, i.e., for every $q \in Q$, $i \in \Sigma^I$, and $o \in \Sigma^O$, there exists precisely one $q' \in Q \cup \{\bot\}$ such that $(q', i) \in \delta(q, o)$.*

The definition of run trees for deterministic safety tree automata is the same as for UCTs. We call all run trees for $\mathcal{A}$ accepting on which $\bot$ is never visited along along any branch.

Note that we have defined determinism a bit different as for word automata here. We introduced a symbol $\bot$ representing that no normal state shall be visited, whereas for word automata, we have one or zero transitions for every state/input symbol combination. This is only for technical convenience, and both approaches can be seen as being equivalent.

Lemma 1. *Given a UCT $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0, \mathcal{F})$ and some bound $b \in \mathbb{N}$, we can build a deterministic safety tree automaton (DST) $\mathcal{A}' = (Q', \Sigma^I, \Sigma^O, \delta', q_0', \mathcal{F}')$ with $|Q'| = (b+2)^{|Q|}$ such that $\mathcal{A}'$ accepts precisely the trees for which the corresponding (unique) run trees for $\mathcal{A}$ visit rejecting states at most b times.*

Proof. The DST $\mathcal{A}'$ can be built as follows:

- $Q' = Q \to \{0, 1, \ldots, b-1, b, \infty\}$
- $\delta(S, o) = \{(f(S, i, o), i) \mid i \in \Sigma^I\}$ for all $S \in Q'$, $o \in \Sigma^O$ for the function $f : Q' \times \Sigma^I \times \Sigma^O \to (Q' \cup \bot)$ with:

$$f(S, i, o) = \begin{cases} \{q' \mapsto \min_{q \in Q} g(q, i, o, q', S(q)) \mid q' \in Q\} & \text{if defined} \\ \bot & \text{otherwise} \end{cases}$$

  for $g : Q \times \Sigma^I \times \Sigma^O \times Q \times (\mathbb{N} \cup \{\infty\}) \to (\mathbb{N} \cup \{\infty\})$ with:

$$g(q, i, o, q', c) = \begin{cases} c & \text{if } (q', i) \in \delta(q, o) \text{ and } q' \notin \mathcal{F} \\ c - 1 & \text{if } (q', i) \in \delta(q, o) \text{ and } q' \in \mathcal{F} \\ \infty & \text{otherwise} \end{cases}$$

- $q_0' = \{q_0 \mapsto b\} \cup \{Q \backslash \{q_0\} \mapsto \infty\}$

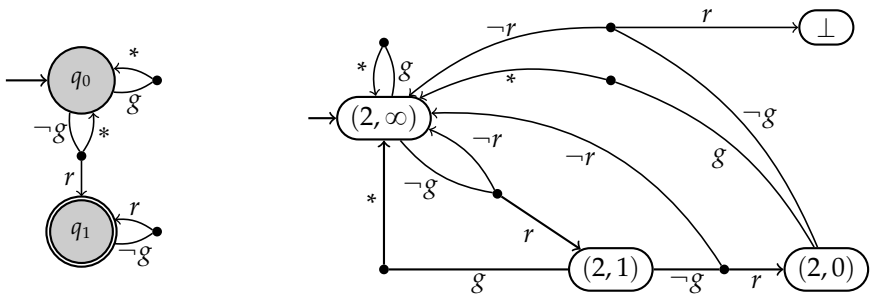

Figure 5.6: Example transformation from a UCT (on the left) to a DST (on the right) using a bound of 2. We depict the edges in the tree automata using little dots, and the label along the line to the dot tells us to which elements of Σ^O the edge refers to. The input/next state combinations are represented by the outgoing edges of the little dot. Along a run tree for the UCT, we are always in the state q_0 for every branch in the respective computation tree to be accepted. Given a branch of the computation tree, we can always see from the corresponding branch in the DST in which states in the UCT run tree we can be along this computation tree branch, and how often we may still visit rejecting states until the bound is reached. The state labels in the DST describe this counter for q_0 and q_1 (in this order). We can see that the counter for q_1 can decrease over time when we stay in q_1 for some time, and that if at some point the counter would drop below 0, the transition leads to $\perp$ instead.

In the DST, the states keep track of the prefix run tree of the UCT for the input observed so far. For example, given a computation tree $\langle T, \tau \rangle$, the corresponding run tree of the UCT $\langle T_r^U, \tau_r^U \rangle$, and the corresponding run tree of the DST $\langle T_r^D, \tau_r^D \rangle$, for some input $t \in (\Sigma^I)^*$ and the unique $t^D \in T_r^D$ with $\tau_r^D(t^D) = S \times t$ for some $S \in Q'$, we have that S maps precisely the states $q \in Q$ to some value other than ∞ for which there exists some node $t^U \in T_r^U$ with $\tau_r^U(t^U) = (q, t)$. At the same time, the definition of δ' ensures that S maps every state of Q onto the number of visits to rejecting states that may occur at most along a path from any node t^U with $\tau_r^U(t^U) = (q, t)$ in order not to violate the requirement that only b visits to rejecting states may occur in total along every branch in the run tree. □

Figure 5.6 explains and depicts the idea of this lemma further. A full proof of correctness for the construction can be found in the original work by Schewe and Finkbeiner (2007). It remains to be shown how we can constructively check a deterministic safety tree automaton for emptiness. We will explain this by drawing a connection to games in the next section.

5.5 Deterministic tree automata and games

Let us now have a look at how to check a (deterministic) tree automaton for emptiness. On a technical level, we can do so by reducing the emptiness problem to game solving. For deterministic safety tree automata, this process results in a safety game, and we have seen already in Chapter 4.2.2 how to deal with these. This fact immediately raises the question why we deal with tree automata at all and not simply go from word automata to games directly in synthesis. The reason is that conceptually, tree automata are richer: for example, universal branching as we have in UCTs does not have a counter-part in the game world. On the other hand, the reformulation of the acceptance problem of a tree automaton as a game clears our view from the details of the automata. Also, by viewing the problem as a game, we can connect it to a whole area of computer science that deals with solving such games, which allows us to import results from this area.

Definition 17. *Given a deterministic safety tree automaton $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0)$, we define its corresponding synthesis game to be $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ with player 0 being the safety player and:*

$$\begin{aligned} V^0 &= Q \\ V^1 &= Q \times \Sigma^O \\ \Sigma^0 &= \Sigma^O \end{aligned}$$

$$\begin{aligned}
\Sigma^1 &= \Sigma^I \\
E^0 &= \{(q,x) \mapsto (q,x) \mid q \in Q, x \in \Sigma^O\} \\
E^1 &= \{((q,x),y) \mapsto q' \mid q \in Q, x \in \Sigma^O, y \in \Sigma^I, (q',y) \in \delta(q,x)\} \\
v_{in} &= q_0
\end{aligned}$$

Theorem 5. *Given a deterministic tree automaton $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_0)$ and its corresponding synthesis game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$, if and only if there exists a winning strategy for player 0 in $\mathcal{G}$, then $\mathcal{A}$ has a non-empty language. Furthermore, strategies $f : (\Sigma^0 \times \Sigma^1)^* \to \Sigma^0$ for player 0 to win $\mathcal{G}$ can be translated to computation trees $\langle (\Sigma^I)^*, \tau \rangle$ by taking $\tau(\epsilon) = f(\epsilon)$ and setting $\tau(t) = f(\tau(\epsilon)t_0\tau(t_0)t_1\tau(t_0t_1)t_2 \dots \tau(t_0 \dots t_n))$ for all $t = t_0 \dots t_n \in (\Sigma^I)^*$.*

In a sense, a safety game obtained from a deterministic safety tree automaton is only a syntactic reformulation of the tree automaton. Note that in our example in Chapter 4.2.2, we had that player 1 represented the system player and was also the safety player. If we take a tree automaton and translate it to its its corresponding synthesis game, we obtain a game in which the system player that produces the outputs is player 0 and is the safety player, while player 1 chooses the inputs and is the reachability player.

Thus, the roles of the two players have been swapped, but it is still the system player who has the safety objective. When taking into account the aim of the synthesis game, this makes sense – we want the system player to ensure that some specification holds for an indefinite amount of time, which for a specification that ensures that "nothing bad ever happens" can only be done with a safety winning condition. However, the construction of the synthesis game now always puts the starting player (namely player 0) into the role of the system player, so the construction is in some sense of "Moore-type". From a technical point of view, this fact only reflects the property of tree automata to look at the labelling of the initial node of a computation tree – thus, tree automata are in a sense also Moore-type affine. We have seen in Definition 15 that we can also spread the language represented by a (universal) word automaton into a tree automaton in a Mealy-type fashion, at the cost of multiplying the number of states by the size of the input alphabet. Consequently applying the construction from Definition 17 leads to another multiplication (of vertices of player 1), this time by the size of the output alphabet. Having both multiplications at the same time can be avoided in practice by making a direct construction from word automata to games in the Mealy setting, possibly by applying the bounded synthesis construction from Lemma 1 on the fly. For Moore semantics, tree automata are a more natural model, and no unnecessary blowup occurs.

Let us conclude this section by discussing the usage of tree automata and games in the synthesis literature. Many publications on synthesis that use Mealy-type semantics do not explicitly deal with tree automata but perform a direct construction from automata to games, and thus avoid this unnecessary blow-up glitch in the theory. Theoretical papers on the other hand often use the Moore-type semantics, as it fits more nicely with the definition of tree automata. Note, however, that Mealy-type synthesis papers that skip tree automata are implicitly still using their main idea: to have an automaton type that can check all branches of a computation tree at the same time.

5.6 Completing the picture: a complete algorithm for synthesis

In this chapter, we have discussed all components of a synthesis approach:

- a word automaton model for linear-time specifications,
- a corresponding tree automaton model (universal co-Büchi tree automata) for the spread of a linear-time specification
- a way to make dealing with universal co-Büchi tree automata more manageable (*bounded synthesis*) by translating them (together with a bound) to deterministic safety tree automata,
- and a way to employ safety games as decision backend.

What remains to be discussed is how we can translate the positional strategies (that we obtain whenever a safety game is winning for the system player) to a Mealy or Moore automaton, and how to make the overall approach complete – recall that for the bounded synthesis approach, we impose a bound of the

number of visits to rejecting co-Büchi states along every branch in a run tree of the universal co-Büchi tree automaton. Thus, for synthesis in general, we want to get rid of the restriction of finding only solutions that adhere to this bound.

From positional strategies to Mealy and Moore machines For translating a winning positional strategy in a synthesis game to a Mealy or Moore machine, we in some sense have to apply the definitions backwards, i.e., translate the positional winning strategy for the safety player back to a regular computation tree that is accepted by the deterministic safety tree automaton from which we built the game, and then build a Mealy or Moore machine with the elements of the equivalence relation of the computation tree as state set.

For simplicity, let us discuss a direct construction here.

Definition 18. *Let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ be a game with player 0 being the safety player built according to the construction from Definition 17 (which ensures that $V^1 = V^0 \times \Sigma^0$), B be the set of bad positions for the safety player with $v_{in} \notin B$, and $f : V^0 \to \Sigma^0$ be a winning positional strategy for player 0. We define the* induced Mealy machine *of f by $\mathcal{M} = (S, \Sigma^0, \Sigma^1, \delta, s_0)$ with:*

$$\begin{aligned} S &= V^1 \\ s_0 &= \text{any position in } \{v_{in}\} \times \Sigma^0 \text{ that is not in } B \\ \delta(s,x) &= (E^0(E^1(s,x), f(E^1(s,x))), f(E^1(s,x))) \\ &\quad \text{for all } s \in S, x \in \Sigma^1 \end{aligned}$$

Likewise, let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{in})$ be a game with player 0 being the safety player, B be the set of bad positions for the safety player with $v_{in} \notin B$, and $f : V^0 \to \Sigma^0$ be a winning positional strategy for player 0. We define the induced Moore machine *of f by $\mathcal{M} = (S, \Sigma^1, \Sigma^0, \delta, s_0, L)$ with:*

$$\begin{aligned} S &= V^0 \\ s_0 &= v_{in} \\ \delta(s,x) &= E^1(E^0(s, L(s)), x) \\ &\quad \text{for all } s \in S, x \in \Sigma^1 \\ L(s) &= f(s) \text{ for all } s \in S \end{aligned}$$

It can be seen that both in the Mealy- as well as Moore-case, the game position set for one player is used as set of states of the automaton. In a sense, the machines computed thus track the position during a play in the game in order to compute the next output and the next state.

Complete bounded synthesis The bounded synthesis approach discussed so far has the disadvantage that it only finds implementations for which their the number of visits to rejecting states in their run trees does not exceed the specified bound. We can solve the problem that we do not know the smallest bound that can be imposed while not ruling out all possible implementations by successively increasing the bound value while repeatedly applying the bounded synthesis procedure. In this way, we would start with a bound of 1 and perform the synthesis process with this bound. If the safety game is found to be losing for the system player, we continue with a bound of 2. If again the process is losing for the system player, we increase the bound again, and so on. Whenever there exists a finite-state implementation for a specification, we will find it in this way, as for every finite-state system that satisfies some specification, there is some bound value that suffices to be accepted by the safety tree automaton that is built in the bounded synthesis process.

We will establish the fact that we can never have a specification for which only infinite-state implementations exist in Chapter 6.1.1. At the same time, we will also show that the bound can be restricted to a value that is exponential in the number of states in the universal co-Büchi automaton such that whenever there is an implementation for a specification, then we find one that adheres to this bound. In a sense, this bound can be called the *worst-case bound* value that we have to consider.

By just starting with the worst-case bound, this completes the method outlined in this chapter to a full decision procedure for the *realisability* of a specification, i.e., for deciding whether there exists an implementation. At the same time, whenever there exists an implementation, we also obtain a Mealy- or Moore-machine representation of it.

Note that from the overall synthesis procedure, we obtain as a corollary that there is no specification that requires non-determinism *in the implementation* in order to be realisable. This follows from the fact that we could represent such a non-deterministic implementation as a set of possible run trees of the system, and our UCT would have to accept all of them for the implementation to satisfy the specification. But then, we could simply pick one of them as our implementation, and we would obtain a deterministic implementation that satisfies the specification as well.

Chapter 6

Advanced topics

We conclude this introduction to reactive synthesis with a short glimpse at a couple of additional concepts that are frequently used in the synthesis literature. The selection of topics has been done according what concepts are needed in the following parts of this thesis.

6.1 More on deterministic automata: parity acceptance and parity games

In the bounded synthesis approach that was outlined in the previous chapter, a key element was the simplification of the acceptance condition and the branching mode of the tree automaton: we started with a universal co-Büchi tree automaton, and translated it to a deterministic safety tree automaton. A safety tree automaton can then be easily translated to a game. However, in this process, we had to introduce a bound in order to make this possible. We have also seen that we can take deterministic word automata and spread them to games, or equivalently, deterministic tree automata, without the need for a bound. However, for this route, the Büchi acceptance/winning condition was not expressive enough to allow for full coverage of all LTL properties.

There is a way to get rid of the necessity to introduce a bound, but still obtain a deterministic tree automaton for synthesis that can subsequently be transformed to a deterministic game. The new concept however comes at a price: we need to trade in the simplicity of the safety (or co-Büchi) acceptance condition and use an acceptance condition that is more complex, namely *parity acceptance*.

The parity acceptance condition generalises the co-Büchi and Büchi acceptance conditions that we have seen previously. Instead of giving a set of accepting states $\mathcal{F}$, as in the Büchi acceptance condition, or giving a set of rejecting states $\mathcal{F}$, as in the co-Büchi acceptance condition, a word or tree automaton with a parity acceptance condition (which are then called *parity automata*) has a *colouring function* that assigns to each state a natural number, called the *colour* of the state. For a run to be accepting in a parity word automaton, we need to have that the highest colour occurring infinitely often along the states of the run is even. So as in the co-Büchi and Büchi acceptance condition, we check what happens infinitely often along a run. Likewise, a run tree is accepting for a parity tree automaton if along every infinite branch in the run tree, the highest colour occurring infinitely often is even.

We start by introducing deterministic parity word automata and sketching how they can be spread to deterministic parity games or deterministic parity tree automata. Then, we discuss shortly how parity automata can be obtained from specifications, and how parity games can be solved (and what important properties they have).

Figure 6.1 shows an example automaton for the LTL specification $(\mathsf{GF}r \wedge \mathsf{GF}g) \vee (\mathsf{FG}\neg r \wedge \mathsf{FG}\neg g)$ (that we used earlier to show the limitations of spreading a non-deterministic Büchi word automaton to a Büchi game). The bottom-most four states are waiting states in which a run of the automaton remains until both g and r have been given (in that order). If both happens infinitely often, then the top state is visited infinitely often, hence the word is accepted, as colour 2 is then the highest one visited infinitely often. If at some point, the r signal or g signals are never given again, the run stays in the bottom four states, and states with colour 1 are visited infinitely often if not at some point the run stays in one of the colour-0 states forever, which happens if and only if the run ends with $\overline{rg}$, i.e., the word satisfies the LTL property $\mathsf{FG}\neg r \wedge \mathsf{FG}\neg g$. Let us now formalise the notion of a parity automaton over words.

Definition 19. *A parity word automaton is defined as a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ with the set of states Q, the alphabet Σ, the transition function $\delta : Q \times \Sigma \to 2^Q$, the initial state $q_0 \in Q$, and the colouring function $c : Q \to \mathbb{N}$. Their run definition is the same as for Büchi word automata, except that a run $\pi = \pi_0\pi_1 \ldots$ is called accepting if and only if the highest colour visited infinitely often along it is even, i.e.,* $\max(\{c' \in \mathbb{N} \mid \forall i \in \mathbb{N} : \exists j > i : c(\pi_j) = c'\})$

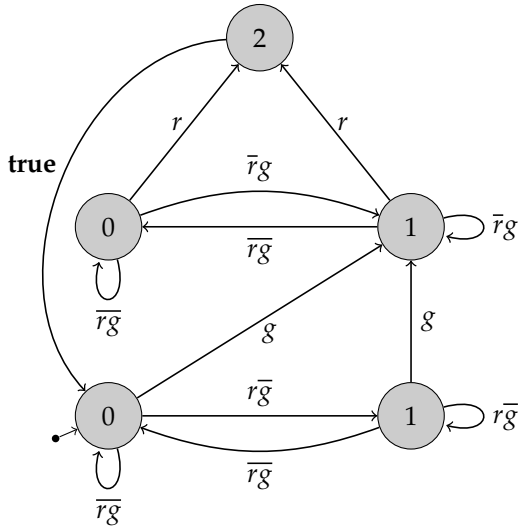

Figure 6.1: A deterministic parity automaton for the specification $(\mathsf{GF}r \wedge \mathsf{GF}g) \vee (\mathsf{FG}\neg r \wedge \mathsf{FG}\neg g)$

is even. The concepts of language equality, determinism, etc., are defined in the same way as for Büchi word automata.

Translating an LTL property or a non-deterministic Büchi automaton to a deterministic parity word automaton is a classical topic in the theory of automata over infinite words (Thomas, 1994; Piterman, 2007). We will not go into the details of a suitable construction, but just rather state one of the tightest known results, given the fact that comprehensive literature dealing only with this topic is available.

Theorem 6 (Piterman, 2007, Theorem 3.10)**.** *For every non-deterministic Büchi word automaton with n states, there exists an equivalent deterministic parity word automaton with* $2n^n n!$ *states and a highest colour of* $2n$*.*

The result by Piterman (2007) is constructive, i.e., he also describes how to build such an automaton in time linear in the worst-case parity automaton size.

A deterministic parity word automaton can be spread to a *deterministic parity tree automaton* $\mathcal{A}' = (Q', \Sigma^I, \Sigma^O, \delta', q_0', c')$, which is done in the same way as for deterministic safety or co-Büchi tree automata, except that we need to take care of the modified winning condition in Definition 14 and Definition 15. The tree automaton then accepts all computation trees for which for the unique run tree, along any branch, the highest colour visited infinitely often is even. The tree automaton can then be translated to a parity game like in Definition 17. Parity games are defined just as Büchi games, but with a modified selection of which plays are winning (namely the one along which the highest colour visited infinitely often is even). Likewise, we can also directly spread a parity word automaton to a parity game as in Definition 9.

The parity games we then get have properties that are similar to the safety and Büchi games we have seen before. In particular, we distinguish the set of positions into the ones from which if starting there, player 0 wins, and the set of positions from which player 1 wins. Also, we only need to consider *memoryless* (i.e., *positional*) strategies, as whenever a player wins a game, then the same player can also win with a positional strategy. This fact directly imposes a size bound on the implementations that we synthesise: we will never obtain a Mealy or Moore machine with more states than the number of the system player's states in the parity game.

6.1.1 Looking back to bounded synthesis

The fact that parity games have memoryless strategies, and that thus an implementation that realises some specification that is encoded into the game does not need more states than the number of positions in the parity game, has consequences to the bounded synthesis approach.

Let us take an LTL specification with n operators. We can translate it to a universal co-Büchi automaton of size $O(3^n)$ using the Miyaho-Hayashi construction from Chapter 3.3. Then, we can apply Theorem 6 to obtain a deterministic parity automaton of size doubly-exponential in n. While Theorem Theorem 6

is actually concerned with non-deterministic Büchi word automata rather than universal co-Büchi word automata (UCW), this imposes no problem, as these are dual. Thus, we can simply apply it to the UCW, and complement the resulting deterministic parity automaton by adding 1 to every of its colours.

The game spread from the resulting parity automaton now has a winning strategy for the system player if and only if the specification is realisable, and if there is a winning strategy, then there is a positional one. All in all, we obtain a doubly-exponential size bound on an implementation for the specification.

This fact finally provides us with an upper limit on the bound values in the bounded synthesis process that we might need to consider. If an implementation of size m is accepted by a UCT with m' states, then along any branch in the run tee of the implementation and the UCT, we might visit rejecting states at most $m \cdot m'$ times, as otherwise we witness a loop that we can repeat infinitely often, leading to the implementation being rejected. Thus, a bound that is doubly-exponential in the number of operators of our LTL specification and linear in the number of UCT states suffices to obtain completeness of the bounded synthesis approach.

Corollary 1 (Schewe and Finkbeiner, 2007)**.** *Given a universal co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_{init}, \mathcal{F})$ over the alphabet $\Sigma_I \times \Sigma_O$ with n states, there exists a Mealy or Moore machine over Σ_I/Σ_O realising $\mathcal{A}$ if and only if there exists one for which the maximum number of visits to rejecting states in $\mathcal{A}$ for some run and some word in the language of $\mathcal{M}$ is less than or equal to $|\mathcal{F}| \cdot 2n^n n!$.*

6.2 On the complexity of synthesis and why the problem is intuitively complicated

Both of the synthesis approaches that we have seen so far have doubly-exponential time and space complexities. In bounded synthesis, we must consider a doubly-exponential bound to achieve completeness, and using the path through parity games, the parity games that we obtain are already of doubly-exponential size. This raises the question if there is no way to do better than this.

Unfortunately, this is not the case. It can be shown that the synthesis problem itself is indeed of doubly-exponential complexity (Pnueli and Rosner, 1989a; Vardi and Wolper, 1986). The proof is lengthy and will not be given in its entirety. Its main idea is to show that we can encode the acceptance of an alternating exponential-space bounded Turing machine of a given word into the LTL synthesis problem (with polynomial blowup in the problem representation). As Chandra et al. (1981) showed, this space complexity class is equivalent to 2EXPTIME. Encoding such a Turing machine acceptance can be done by writing an LTL specification for a system that prints out the Turing tape contents along a computation of the Turing machine in a sequential manner, for which LTL is sufficiently expressive. We encode the possible transitions of the Turing machine in the specification, and simulate the alternation by using the input to the system.

This insight also tells us something about the game constructions that we discussed earlier in Chapter 4.2. We have seen there that one cannot simply translate a non-deterministic Büchi automaton to a game by splitting up the transitions and obtain a game which has a winning strategy for the system player if and only if there exists an implementation for the specification described in the automaton. However, we did not know at that point whether we just tried to do this in the wrong way. Now that we have established the complexity of the synthesis problem, we get as a corollary that there is no simple way (with only polynomial blowup in the translation) to fix the translation from non-deterministic Büchi automata to Büchi games. If such a translation existed, the overall synthesis approach would only have exponential complexity, which cannot be the case, as it has been proven that EXPTIME≠2EXPTIME (see, e.g., Seiferas et al., 1978). Recall that the incompleteness of a synthesis approach that works by spreading a non-deterministic automaton to a game is caused by the fact that the system player in the Büchi game has to guess in advance which transition in the automaton that the game is built from is to be taken in order to make the play winning. We can conclude that it is this inability of the system player to guess this ahead of time that is causing a complexity increase of one exponent.

6.3 Beyond linear-time: alternating tree automata

After we have discussed two synthesis flows in the previous chapters (one based on bounded synthesis, and one based on deterministic parity automata and parity games), let us now turn our attention to an extension of linear-time synthesis.

So far, we have assumed to deal with linear-time specifications. In model checking, branching-time specification logics such as computation tree logic (CTL, Emerson and Clarke, 1982) are also a popular specification formalism. For synthesis, branching-time specifications are more seldomly used, as the additional expressivity is not always useful. For example, in CTL, we could state for a mutual-exclusion (mutex) protocol that there should be some path in the run tree of the system that we are trying to synthesise on which grants are obtained infinitely often. Whenever we also want to specify *under which conditions* this should happen (here, if requests are issued infinitely often), then linear-time specifications are however sufficient again to state the specification. Nevertheless, in some applications, branching-time logics are useful. Emerson and Clarke (1982) originally introduced CTL to synthesise *synchronization skeletons* that are meant to be refined with actual code in a system engineering process. Here, we might want to be a able to state that there should be a way to make some transitions in the system, as the actual condition under which the transition should be taken is only determined during the later refinement.

To accommodate branching-time specifications in a synthesis process, we need a richer tree automaton model. For automata over infinite words, we have seen non-deterministic, deterministic, universal, and alternating automata so far. For trees, our tree automaton models either had universal or deterministic branching. In order to accommodate branching-time specifications, we need to use non-determinism in tree automata as well, and either apply a non-deterministic or alternating branching mode.

Definition 20. *An alternating Rabin tree automaton is defined as tuple* $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_{init}, \mathcal{F})$ *with the set of states* Q*, the set of directions* Σ^I*, the set of labels* Σ^O*, the transition function* $\delta : Q \times \Sigma^O \to \mathcal{B}(Q \times \Sigma^I)$*, the initial state* q_{init}*, and the set of* acceptance pairs $\mathcal{F} \subseteq (2^Q \times 2^Q)$.

Let $\langle T, \tau \rangle$ *be a computation tree (of a system) with* $T = (\Sigma^I)^*$ *and* $\tau : T \to \Sigma^O$*. We call a tree* $\langle T_r, \tau_r \rangle$ *a run tree of* $\mathcal{A}$ *and* $\langle T, \tau \rangle$ *iff:*

- $T_r \subseteq \mathbb{N}^*$, $\tau_r : T_r \to Q \times (\Sigma^I)^*$
- $\epsilon \in T_r$ *and* $\tau_r(\epsilon) = (q_{init}, \epsilon)$
- *For every* $t \in T_r$ *with* $\tau_r(t) = (q, t')$ *for some* q *and* t' *and* $x \in \mathbb{N}$*, if* $tx \in T_r$*, then* $\tau_r(tx) = (q', t'x')$ *for some* $q' \in Q$ *and* $x' \in \Sigma^I$*.*
- *For every* $t \in T_r$ *with* $\tau_r(t) = (q, t')$ *for some* q *and* t'*, let* $S = \{tx \in T_r \mid x \in \mathbb{N}\}$ *be the set of* t*'s successors in the run tree, and* $R = \{\tau(t'') \mid t'' \in S\}$ *be the set of its labels. We translate* R *to a characteristic function* $f : (Q \times \Sigma^I) \to \mathbb{B}$ *over* Q *by setting* $f(q', x) =$ **true** *if* $(q', t'x) \in R$ *and* $f(q', x) =$ **false** *otherwise (for all* q' *and* x*). For the tree to be valid, we require that* f *satisfies the transition function* δ *from* q *for the character to be read next, i.e.,* $f \models \delta(q, \tau(t'))$.

A run tree is accepting if every of its infinite branches is accepting. The automaton accepts all computation trees that have accepting run trees.

Whether a branch is accepting depends on the Rabin acceptance condition *of the tree automaton. For determining the acceptance, we look at the states visited infinitely often along the branch. Formally, for some branch* $t = t_0 t_1 t_2 \in \mathbb{N}^\omega$ *such that for all* $i \in \mathbb{N}$*,* $t_0 t_1 \dots t_i \in T_r$*, we define* $\inf_Q(t) = \{q \in Q \mid \forall i \in \mathbb{N} \exists j > i : \tau_r(t_0 \dots t_j) = (q, t')$ *for some* $t' \in T\}$*. The branch* t *is accepting if there exists some* $(E, F) \in \mathcal{F}$ *such that* $\inf_Q(t) \cap E = \varnothing$ *and* $\inf_Q(t) \cap F \neq \varnothing$.

Let us examine the expressiveness of alternating tree automata by means of an example.

Example 17. Let $\Sigma^I = \{a\}$, $\Sigma^O = \{b\}$, and $\mathcal{A} = (Q, \Sigma^I, \Sigma^O, \delta, q_{init}, \mathcal{F})$ be an alternating tree automaton with $Q = \{q_{init}\}$, $\mathcal{F} = \{(Q, \varnothing)\}$, $\delta(q_{init}, \{b\}) =$ **true**, and $\delta(q_{init}, \varnothing) = (q_{init}, \varnothing) \vee (q_{init}, \{a\})$.

This automaton accepts all computation trees on which some node is labelled by $\{b\}$. To see this, consider a tree with some node having this label. We start from the root node by sending state q towards this node. Once it is reached, the transition function maps to **true** at this point. On the other hand, if there is no node labelled by $\{b\}$ in the run tree, then every run tree has an infinite branch, and since for the only acceptance pair (E, F) in $\mathcal{F}$, we have $E = Q$, no infinite branch t in the run tree can be accepting, so the computation tree is rejected. ★

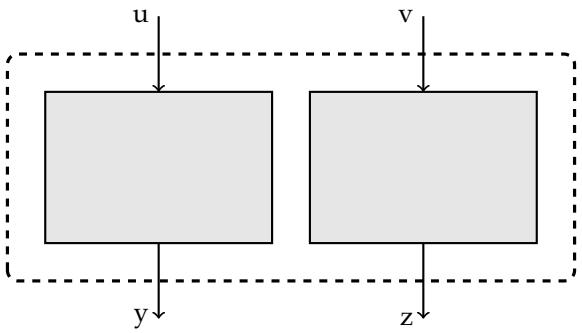

Figure 6.2: The A0 architecture as defined by Pnueli and Rosner (1990)

We will not introduce computation tree logic (CTL) and its extension CTL* here, but rather just state that alternating tree automata offer sufficient expressiveness to cover both logics. For CTL*, Kupferman et al. (2000) describe a way to translate a CTL* formula into a so-called *weak hesitant alternating automaton*. These can be translated to alternating Rabin tree automata, which we can then check for emptiness in order to perform synthesis. The interested reader is referred to the work by Kupferman and Vardi (1997) for more details.

6.4 Distributed synthesis

Many systems in practice are not monolithic but rather consist of multiple sub-systems that work together in a distributed fashion in order to complete the overall task. If we want to synthesise such systems, then we will need to adapt our synthesis methodology to capture such settings.

Formally, in distributed synthesis, we do not any longer provide a temporal specification and an interface to the synthesis tool, but rather a temporal specification and an *architecture*. The architecture describes the components of the system to synthesise, i.e., which inputs and outputs they have, and how they are connected.

Figure 6.2 shows an example architecture (called the *A0 architecture* by Pnueli and Rosner, 1990). Here, we have two processes. The first process reads the input signal u und writes to the output signal y. The second process on the other hand reads v and writes to z. A specification in this setting is a temporal logic formula over $\mathsf{AP} = \{u, v, y, z\}$. Apparently, the left process cannot read input v and the right process cannot read input u. Intuitively, this is where distributed synthesis gets complicated: we have to synthesise processes that do not always have access to the full input to the system, and the overall task that the system has to fulfil must be orchestrated in a way that ensures that the individual processes do not need more information than they have to do their share of the overall task.

The processes themselves are Mealy or Moore machines that run in parallel at the same time. Semantically, we can describes their joint behaviour by building a *product machine* of the two processes. Let us discuss this by means of an example.

Definition 21. *Let $\mathcal{M}_1 = (S_1, \Sigma_1^I, \Sigma_1^O, \delta_1, s_{init,1}, L_1)$ and $\mathcal{M}_2 = (S_2, \Sigma_2^I, \Sigma_2^O, \delta_2, s_{init,2}, L_2)$ be Moore automata. We define the synchronous product of $\mathcal{M}_1$ and $\mathcal{M}_2$ as the Moore automaton $\mathcal{M}' = (S', \Sigma'^I, \Sigma'^O, \delta', s'_{init}, L')$ with:*

- $S' = S_1 \times S_2$
- $\Sigma'^I = \Sigma_1^I \times \Sigma_2^I$
- $\Sigma'^O = \Sigma_1^O \times \Sigma_2^O$
- *For all $(s_1, s_2) \in S'$ and $(x, x') \in \Sigma_1^I \times \Sigma_2^I$, we have $\delta'((s_1, s_2), (x, x')) = (s'_1, s'_2)$ for $\delta^1(s_1, x) = s'_1$ and $\delta^2(s_2, x') = s'_2$*
- $s'_{init} = (s_{init,1}, s_{init,2})$
- *For all $(s_1, s_2) \in S'$, we have $L'(s_1, s_2) = (L_1(s_1), L_2(s_2))$*

In this definition, the systems do not actually interact with each other. We can now apply the run tree definition for a Moore machine to a product automaton and have a definition of the behaviour of such a distributed system. In practice, the actual product definitions can be a bit different, e.g., when we have communication between the processes. Then, depending on the concrete semantics (i.e., if communication between the processes is instantaneous or if it needs one clock cycle), the construction is altered.

As distributed synthesis is intuitively more complicated than synthesising a single process, this raises the question if the complexity of distributed synthesis for LTL specifications is still the same as for LTL synthesis of monolithic systems. Unsurprisingly, it is not. Even worse, for the architecture from Figure 6.2, the synthesis problem is undecidable, i.e., there exists no algorithm that, given some LTL specification over $\mathsf{AP} = \{u, v, y, z\}$, tests if there exist implementations for the two processes that ensure that their synchronous product satisfies the specification. This fact has been shown by Pnueli and Rosner (1990).

The undecidability of distributed synthesis can be proven by reducing the halting problem of an arbitrary Turing machine to the distributed synthesis problem. This means that we give a procedure for translating a (deterministic) Turing machine description to a specification that is realisable in the A0 architecture if and only if the Turing machine eventually halts.

Essentially, the idea in the construction is to translate a Turing machine description into an LTL specification for the overall system that forces the two processes to output the complete Turing machine tape computations (along with their states and tape positions) along the run of the Turing machine through their local outputs.

Let us assume that we have a tape alphabet Γ of size k and the Turing machine states S are of cardinality k'. For every position of the tape, we need to encode an element from the following set into our binary output stream:

$$E = \# \cup (\Gamma \times (S \cup \{\bot\}))$$

The special symbol # shall be used to denote the boundaries of a tape. With it, we can output a sequence of tape contents in a single binary output stream one-after-the-other without having to guess where one tape content ends and the next content starts. The alphabet Γ is used for all other tape positions, and we choose to encode the machine state along with the tape. A value in S means that the Turing machine's head is at some position, while $\bot$ means that it is not. To squeeze these many elements into a single bit output stream, we will need to output $2 \cdot \lceil \log_2 |E| \rceil$ bits sequentially per tape position. The factor of 2 in this equation comes from the added requirement that for the specification of the system to synthesise, we will need to be able to tell from looking at the stream locally whether we are at the beginning of a new element of E in the binary stream or not. Otherwise, we would only need $\lceil \log_2 |E| \rceil$ elements. Readers also interested in the next part of this thesis will see in Chapter 9.1.5 how such an encoding can be made.

The LTL specification consists of several parts, i.e., we have:

$$\psi = \psi^1 \wedge \psi^2 \wedge \psi^3 \wedge \psi^4 \wedge \psi^5,$$

where every ψ^i for $i \in \{1, \ldots, 5\}$ is an LTL formula.

The local inputs u and v of the two processes serve as *starting signal*s. If a process obtains a **true** value in some computation cycle for the first time, then in such a case, it must must output the initial tape configuration and the successor tape configuration of the initial one on its local output signal (in a sequential manner). For the first process, the expression ψ^1 specifies this, while for the second process, it is expression ψ^2. Technically, we can write ψ^1 as $(\mathsf{G}\neg u \vee (\neg u \,\mathsf{U}((T(0)) \wedge \mathsf{X}(T(1)) \wedge \mathsf{XX}(T(2))\ldots))) \wedge (u \,\mathsf{R}\neg y)$, where T is a sequence of y and $\neg y$ symbols that represent the initial two tape contents and machine states. The expression for ψ^2 is analogous.

Expression ψ^3 now states that whenever at a point in time, both of the tapes put out over the signals y and z are at a boundary point, and the next tape content produced by the right process is the immediate successor configuration of the next tape content produced by the left process, then this should also be the case for the next next tape contents. Note that checking if the y-tape content is a successor of the z-tape content can be made by an Until- or Release-formula, where the two tape contents are compared locally until the tape boundary symbol comes. The fact that the tape boundaries might be pushed along the way is of relevance, but can also be handled locally, as the tape can extend by at most one tape cell.

Expression ψ^4 is again the same as ψ^3, but with the two processes swapped. Expression ψ^5 finally says that eventually, we reach an accepting state of the Turing machine.

The important observation is now that if the product of the two processes satisfies ψ, then they are forced to output the *whole* series of tape contents of the Turing machine by the conjuncts ψ^1 to ψ^4 once they obtain a **true** as input (for the first time). Obviously, by ψ^1 and ψ^2, they must output the initial two tape contents and state configurations. But then, if one of the processes is about to write the second configuration to its signals, it does not know if at the same time, the other process obtains a *go* signal.

Let, without loss of generality, the left process be the one that starts first. It has just put out the first tape content, and then the right process obtains a start signal. It will thus output the first tape content while the first process outputs the second tape content. Note that in this case, the condition in ψ^3 applies, and thus the tape content put out by the left process afterwards has to be the successor of the tape content put out by the right process then. Note that the right process is still forced to output the second configuration. Thus, the left process will need to output the third configuration then. Now the right process will not know that the left process has been started first, but must assume that it has been started at the time after the second process has produced the first tape content. Thus, by the same line of reasoning, after the second tape content, it will have to write the third tape content in the machine's execution to z. By applying the same argument again, we obtain that then the first process also has to output the fourth tape content. By induction over a tape content's position in the computation of the Turing machine, we obtain that indeed both processes have to output the complete computation of the Turing machine.

Note that the conjunct ψ^5 then ensures that ψ is realisable for this architecture if and only if the Turing machine eventually halts. This shows the undecidablity of the distributed synthesis problem for the A0 architecture.

Note that by complementing ψ^5, we obtain a specification that is realisable if and only if the given Turing machine *does not halt*. So this case is still suitable for showing that the distributed synthesis problem is undecidable, but ψ^5 can be written as a safety property this way. As the other parts of ψ are also representable as safety properties, this shows that distributed synthesis is even undecidable for safety specifications.

There are many architectures with an undecidable synthesis problem. Finkbeiner and Schewe (2005) showed that the decidability of the synthesis problem for an architecture depends crucially on the *informedness* of the processes in the architecture. If there are two processes like in the A0 architecture that are incomparably informed with respect to the overall input (the right process cannot read input signal u in that architecture, and the left one cannot read v), then the synthesis problem for the architecture is undecidable. However, if there is a strict hierarchy of the levels of informedness of the processes, synthesis is decidable. A construction for doing so is however beyond the scope of this introduction, and the interested reader is referred to the work by Finkbeiner and Schewe (2005) for details.

Chapter 7

Summary and discussion

This part of the thesis discussed the basic ideas behind today's reactive synthesis approaches. We started by explaining how systems and specifications can be described, and dealt with ways to obtain finitary such descriptions for systems that have infinitely long computation traces.

Then, we worked our way through the automata-theoretic basics for model checking and synthesis. We learned about the important concept of alternation along the way, and have seen a full translation workflow from linear-time temporal logic to non-deterministic Büchi automata. After a quick look at the model checking problem, we introduced the important concept of games, and started with a simple game model, namely safety games. From this game model that was already suitable for the synthesis from safety specifications, we moved to Büchi games as the more expressive framework. We saw that when just spreading a non-deterministic Büchi automaton for a specification to a Büchi game, we do not obtain a complete synthesis approach, however. To solve this problem, we dived into tree automata, and introduced universal tree automata as a tool that is equally useful in model checking and synthesis. By bounding the *reactivity of a solution* from below, we were able to move from there to deterministic safety tree automata, whose emptiness we were then able to test by a reduction to safety games. A winning strategy in these games then represents an implementation that satisfies the original specification.

In the discussion of this *bounded synthesis* workflow (Schewe and Finkbeiner, 2007), it was made sure that *all* of the necessary constructions were given explicitly, and thus this introduction can serve as a complete and self-contained primer to the subject. Nevertheless, the workflow is not the only one that can be found in the literature. To provide a bit of insight into alternatives, we then discussed parity games and how they can be used for synthesis.

We finally augmented the discussion of algorithmic solutions to reactive synthesis from linear-time specifications by some additional topics. We started by giving some intuition on why the simple Büchi game construction from non-deterministic Büchi automaton specifications cannot be complete. Then, we had a look at synthesis from branching-time specifications. Finally, we had a short look at the distributed synthesis problem and in particular the general undecidability of it.

This introduction is by no means exhaustive. For pretty much every aspect of reactive synthesis, there are works that improve, analyse, or replace one or more of the concepts discussed.

Let us conclude this introduction by having a look at some of the assumptions that we implicitly or explicitly made in this discussion of reactive system synthesis.

We assumed a discrete time model in which the world evolves in steps and systems never go out of service. The discrete nature of time in this setting is justified by the fact that many settings can be abstracted in practice in this way, and a large number of systems actually behaves in this way, for example on-chip clocked circuits. Nevertheless, we might want to work with a richer time model to skip this abstraction step. Unfortunately, synthesis becomes a much harder task in this setting. For example, for metric interval temporal logic (MITL), which can be seen as a real-time extension of LTL, the synthesis problem is undecidable (Bouyer et al., 2006). In settings in which a reasonable discrete-time abstraction can be found, discrete-time synthesis is a reasonable way to mitigate this problem. The fact that we assume that our system never goes out of service on the other hand is easily justified by the fact that we never needed something like a maximum running time anywhere in the synthesis process. Furthermore, such a time bound is very hard or even impossible to give for many systems. Even if we had a time bound, it would typically be very high, and even algorithms that run in time linear in the bound would probably be infeasible in practice. Finally, with a time bound, we would lose the possibility to write simple specifications such as $\mathsf{G}(r \rightarrow \mathsf{FX}g)$. If r is and input and g is an output, such a specification would either trivially be true if going out of service would relieve the system from having to ensure that eventually g holds, as then the system might simply wait for the end of the run, or otherwise if a g must then be given before the system goes out of service, getting an r at the last clock cycle would lead to a violation of the specification. So in any case, the specification would be

pointless. Nevertheless, it should be noted that LTL for finite traces is a research topic in the area of runtime verification (see, e.g., Bauer et al., 2010).

We restricted ourselves to deterministic solutions as the correctness of the synthesis constructions implied that such solutions suffice. Furthermore, we only considered non-clairvoyant systems as only these can actually be executed in practice.

The focus on linear-time specifications as basis for synthesis is based on the observation that in practice, specifications often tell us what shall happen under which conditions. For such specifications, branching-time is typically not needed, and branching-time logics with a lower synthesis complexity like computation tree logic (CTL), which is popular in model checking, are often insufficient. For the case of LTL, for example, an LTL property such as $\mathsf{FG}(\mathit{initialised})$ cannot be expressed. Such a property could be part of the specification of a system that has a finite initialisation period that must eventually end. However, it should be noted that for CTL, the reactive synthesis problem is actually EXPTIME-complete (Emerson and Clarke, 1982; Kupferman and Vardi, 1997), and thus simpler.

In model checking, the basic computation model are typically labelled transition systems (LTS). In this introduction, we used Mealy or Moore machines, as these make the distinction between input and output explicit, as needed for synthesis. We could of course also synthesise labelled transition systems under an additional *input-progressiveness restriction* that for every possible input and reachable state, there is a successor state labelled by that input. Such a model would be similar to the Moore machines that we have here, but the non-syntactic nature of the restriction makes dealing with them a bit more complicated.

Finally, as specification logic, we have used LTL, despite the fact that LTL is not equi-expressive to the full *class of ω-regular properties* (i.e., the ones that can be represented as deterministic parity word automata). The reason is mainly that LTL is not only sufficient for many applications, but also a very popular formalism in the recent time. Nevertheless, the bounded synthesis approach that we explained in this introduction can be used for all ω-regular specifications, as universal co-Büchi word automata, which serve as intermediate representation of the specification in the synthesis workflow, are capable of expressing all of these properties.

Part II

Symmetric synthesis

Chapter 8

Symmetric architectures

Intuitively, a distributed reactive system is called *symmetric* if it is defined as a network of processes that all have the same implementation. A key idea in this setting is that the processes cannot distinguish among themselves (Angluin, 1980). Thus, they have no access to their identity and necessarily the same number of input and output signals.

In this chapter, we formalise this idea, and analyse for which settings synthesis of symmetric systems is decidable and for which this is not the case.

We start by defining *symmetric architectures*. These describe how a process is instantiated in order to obtain a full reactive system. Afterwards, we identify undecidable cases and decidable cases. For the latter, we give a suitable algorithm in the next chapter.

8.1 Architecture definition

For defining symmetric architectures, we must first state how the process that we want to instantiate in the architecture looks like. Formally, this is done by stating its *signature*, which is a tuple $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ with the two sets AP^I and AP^O, which denote the *input signals* and *output signals* of the process, respectively. In every computation cycle of the overall system, the process reads the values of its input signals and chooses the values of its output signals.

Definition 22 (Symmetric architecture)**.** *Given an interface $\mathcal{I}$, a* symmetric architecture *over $\mathcal{I}$ is a tuple $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ with:*

- *the set of (internal) signals S,*
- *the* process instantiation set *P,*
- *the* global input signal set *AP^I_g,*
- *the* input edge function *$E^{in} : (P \times \mathsf{AP}^I) \to (S \cup \mathsf{AP}^I_g)$, and*
- *the* output edge function *$E^{out} : (P \times \mathsf{AP}^O) \to S$.*

We discuss the intuition of this definition by means of an example. Figure 8.1 shows an architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ for a process with interface $\mathcal{I} = (\{a, b, c\}, \{d\})$. In the architecture, we have two copies of the same process, named *One* and *Two*. The input signals of the first process are directly connected to the input signals of the overall system, so we have $E^{in}(One, a) = u$, $E^{in}(One, b) = v$, and $E^{in}(One, c) = w$. The output of the first process is read as local input a by process *Two*, so we have $S = \{s_1, z\}$ and $E^{out}(One, d) = s_1 = E^{in}(Two, a)$. The signal z is not read by any process, and thus can be seen as an output-only signal.

For architectures to make sense, it has to be ensured that every signal has precisely one producer.

Definition 23. *Given an architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ for some process interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$, we call $\mathcal{A}$* well-formed *if for all $x \in S$, there is precisely one $(p, y) \in (P \times AP^O)$ such that $E^{out}(p, y) = x$.*

Henceforth, we will only discuss well-formed symmetric architectures. Let us now define the semantics of a symmetric architecture. To simplify the notation in the next chapter, we choose a Moore-type computation model here, such that in every computation cycle, a process first chooses its output, and then obtains its input. This choice does not restrict the generality of our results, as the adaptation of the concepts to follow for a Mealy-type computation model is simple.

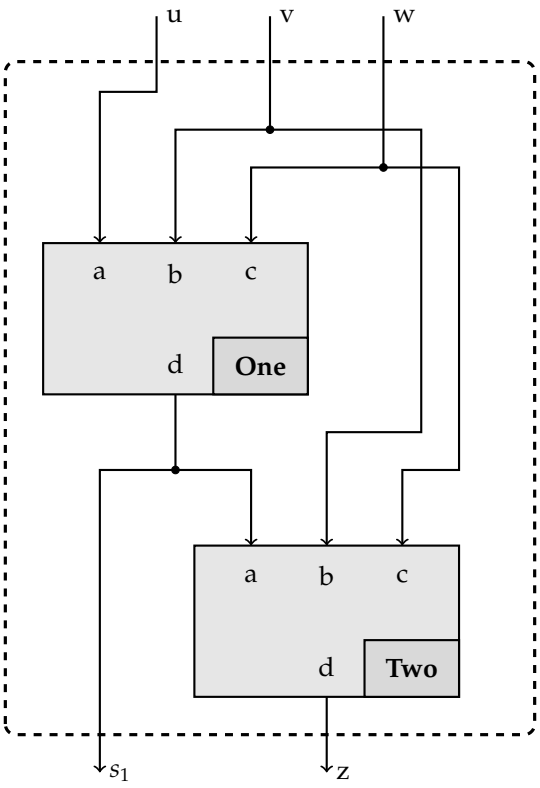

Figure 8.1: An example symmetric architecture. The dashed lines represent the overall system and how it presents itself to its environment.

Definition 24. *Given an architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ for some process interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ and some Moore machine $\mathcal{M} = (Q, I, O, \delta, q_0, L)$ with $I = 2^{\mathsf{AP}^I}$ and $O = 2^{\mathsf{AP}^O}$, we define the* aggregated Moore machine *of the architecture and $\mathcal{M}$ as $\mathcal{M}' = (Q', I', O', \delta', q_0', L')$ with:*

- $Q' = (P \to Q)$,
- $I' = 2^{\mathsf{AP}^I_g}$,
- $O' = 2^S$,
- *for all $f \in Q'$, we have $L'(f) = \{s \in S \mid \exists (p, x) \in P \times \mathsf{AP}^O : E^{out}(p, x) = s,\ x \in L(f(p))\}$,*
- *for all $f \in Q'$ and $X \subseteq \mathsf{AP}^I_g$, $\delta'(f, X) = f'$ such that for all $p \in P$, $f'(p) = \delta(f(p), \{x \in \mathsf{AP}^I \mid E^{in}(p, x) \in (X \uplus L(f))\})$, and*
- *for all $p \in P$, $q_0(p) = q_0$.*

This definition ensures that the values of all signals are "exported" from the aggregated finite-state machine. Thus, when specifying the system behaviour of an aggregated system in a language such as *linear-time temporal logic* (LTL), we can refer to the signals used internally between the components. As there is no necessity to do so, this does not impose a restriction to the generality of our (positive) results, however.

We now have set the scene to define the realisability and synthesis problems for symmetric architectures.

Definition 25. *Given an interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$, a well-formed architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ for $\mathcal{I}$, and a specification ψ in CTL* or LTL over $S \cup \mathsf{AP}^I_g$, or in form of a word or tree automaton over 2^S and $2^{\mathsf{AP}^I_g}$, the symmetric realisability problem for $\mathcal{I}$, $\mathcal{E}$, and ψ is to check if there exists some Moore machine $\mathcal{M}$ over the input 2^{AP^I} and output 2^{AP^O} such that the aggregated Moore machine of $\mathcal{E}$ and $\mathcal{M}$ satisfies ψ. In the symmetric synthesis problem, in case of realisability, a corresponding Moore machine $\mathcal{M}$ is to be computed in addition.*

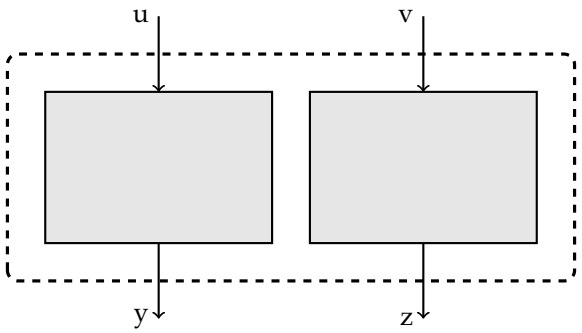

Figure 8.2: The A0 architecture as defined by Pnueli and Rosner (1990)

We call a setting $(\mathcal{I}, \mathcal{E})$ decidable for specifications in CTL, LTL, or as word or tree automata if there exists an algorithm that can decide the symmetric realisability of a specification ψ for the interface $\mathcal{I}$ and the architecture $\mathcal{E}$.*

8.2 Decidable and undecidable architectures

Previous works showed that the distributed synthesis problem is undecidable in many cases (Pnueli and Rosner, 1990; Kupferman and Vardi, 2001a; Finkbeiner and Schewe, 2005). In this problem, we are given some architecture, and we ask the question whether there exist implementations for the processes in the architecture such that the overall distributed system satisfies some property. Finkbeiner and Schewe (2005) proved that the problem is decidable if and only if there exists no *information fork* in the architecture. An information fork is a pair of processes that are incomparably informed, i.e., for which each of the processes has access to some global input that the other process cannot read. If we now add the requirement that the two processes must have the same implementation, as we do in symmetric synthesis, we cannot expect the problem to become easier. In this section, we identify undecidable settings. We focus on characterising the structural properties of architectures that make their symmetric realisability problem undecidable.

The undecidability proofs in the following reduce the following problem to symmetric realisability.

Definition 26 (A0 architecture realisability problem, Pnueli and Rosner, 1990)**.** *Let u and v be two input signals, y and z be two output signals, and ψ be a specification in CTL, LTL or a safety word automaton. The distributed realisability checking problem for the A0 architecture is to determine whether there exists a Moore machine $\mathcal{M}_1$ over the input alphabet $2^{\{u\}}$ and output alphabet $2^{\{y\}}$ and a Moore machine $\mathcal{M}_2$ over the input alphabet $2^{\{v\}}$ and output alphabet $2^{\{z\}}$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ running in parallel realise ψ.*

Figure 8.2 depicts the setting. Note that the description of the synchronous product of two machines, which defines semantically how two machines running in parallel behave, can be found in Definition 21 (on page 71).

Lemma 2 (Pnueli and Rosner, 1990)**.** *The realisability problem for the A0 architecture is undecidable.*

Proof. A proof for this lemma was given by Pnueli and Rosner (1990), and is based on constructing a specification that requires both machines to output the tape content (and state) evolution of a Turing machine on y and z after the computation has been triggered by u and v, respectively. By adding a conjunct that an accepting state shall be visited by the Turing machine, we ensure that the specification is A0-realisable if and only if the Turing machine does eventually halt. The specification can be encoded as a CTL or LTL formula.

The original proof by Pnueli and Rosner (1990) was not concerned with pure safety specifications. It can however be modified slightly to only require a safety specification. This is done by altering the specification such that an accepting state shall *never* be visited by the Turing machine. This complements the result of a realisability check, but preserves undecidability. The resulting specification can then be represented as a deterministic safety word automaton. □

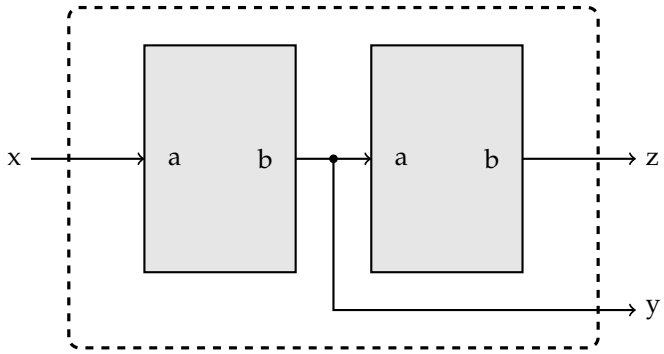

Figure 8.3: Graphical representation of the S0 symmetric architecture.

8.2.1 Internal signals

Consider the architecture depicted in Figure 8.3. Henceforth, we call this architecture the *S0 architecture*. It consists of two copies of a process with a single input bit and a single output bit. In a sense, this architecture is the simplest one for which the output of one process is used as input to another process. We show in this subsection that this symmetric architecture is undecidable. From this result, the undecidability for many other settings that incorporate this architecture as a sub-architecture follows.

We start by explaining a key component for all of the undecidability proofs to follow. The concept of *signal and specification compression* allows us to time-share many signals into one, and translate a specification over many signals into an equivalent specification for a single signal.

Definition 27. *Let* AP *be a set of signals. We call a function* $f : (2^{\mathsf{AP}})^\omega \to (2^{\{\chi\}})^\omega$ *for some Boolean variable* χ *a* compression function *if* f *is injective.*

We call a function f' *that maps a specification (in some form) over the signal set* AP *to a different specification over the signal set* $\{\chi\}$ *the* adjunct compression function *to* f *if for all* $w \in (2^{\mathsf{AP}})^\omega$ *and specifications* ψ *over* AP*, we have that* $w \models \psi$ *if and only if* $f(w) \models f'(\psi)$*.*

There exist pairs (f, f') of compression functions for arbitrary signals sets AP and any linear-time specification formalism that we consider here (LTL and all word automaton types discussed in this thesis) such that a specification only grows polynomially when applying f'. We will prove this fact for the example of LTL. The other cases work accordingly.

Definition 28. *Let* AP *be some set of signals, which, w.l.o.g, we assume to be* $\{x_1, \ldots, x_n\}$*. We define a compression function* f^{LTL} *over* AP *as follows: for every* $w = w_0 w_1 w_2 \ldots \in (2^{\mathsf{AP}})^\omega$*, we set* $f^{\mathrm{LTL}}(w) = w'_0 w'_1 w'_2 w'_3 \ldots$ *such that:*

- $\forall i \in \mathbb{N}$, $w'_{2i(n+1)} = w'_{2i(n+1)+1} = \{\chi\}$,
- $\forall i \in \mathbb{N}$ *and* $j \in \{1, \ldots, n\}$*, we have* $w'_{2i(n+1)+2j} = \varnothing$*, and*
- $\forall i \in \mathbb{N}$ *and* $j \in \{1, \ldots, n\}$*, we have* $\chi \in w'_{2i(n+1)+2j+1}$ *if and only if* $x_j \in w_i$*.*

The functioning of the compression function f^{LTL} is exemplified in Figure 8.4. One clock cycle in the four-bit-per-character version of a word is spread to 10 computation cycles in the one-bit-per-character version of the word. Every 10 cycles, the 2-cycle *character start sequence* $\{\chi\}\{\chi\}$ is instantiated, followed by four two-cycle slots for every signal in AP. Note that the construction ensures that whenever we have $\{\chi\}\{\chi\}\varnothing$ as a part in a compressed word, then we know that a character start sequence begins on the first occurrence of $\{\chi\}$ in this part. Our interest in f^{LTL} lies in the fact f has an adjunct specification compression function f'^{LTL} for LTL properties.

Lemma 3. *A specification compression function* f'^{LTL} *that corresponds to* f^{LTL} *can be defined inductively over the structure of an LTL formula as follows (for* $\mathsf{AP} = \{x_1, \ldots, x_n\}$*):*

- *For* $\psi =$ **false**, *we set* $f'^{\mathrm{LTL}}(\psi) =$ **false**. *Likewise, for* $\psi =$ **true**, *we set* $f'^{\mathrm{LTL}}(\psi) =$ **true**.

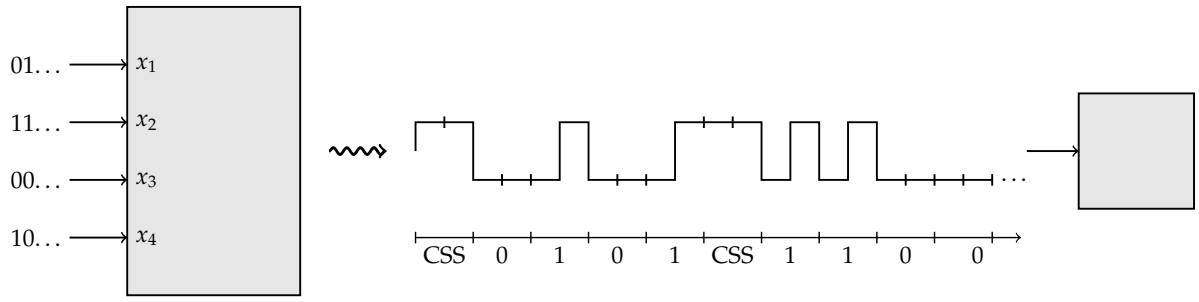

Figure 8.4: An example for compressing a word with $|\mathsf{AP}| = 4$.

- *For $\psi = \psi_1 \wedge \psi_2$, $\psi = \psi_1 \vee \psi_2$, and $\psi = \neg\psi_1$ for some LTL formulas ψ_1 and ψ_2, we have $f'^{\mathrm{LTL}}(\psi) = f'^{\mathrm{LTL}}(\psi_1) \wedge f'^{\mathrm{LTL}}(\psi_2)$, $f'^{\mathrm{LTL}}(\psi) = f'^{\mathrm{LTL}}(\psi_1) \vee f'^{\mathrm{LTL}}(\psi_2)$, and $f'^{\mathrm{LTL}}(\psi) = \neg f'^{\mathrm{LTL}}(\psi_1)$, respectively.*
- *For $\psi = x_j$ for some $x_j \in \mathsf{AP}$, we set $f'^{\mathrm{LTL}}(\psi) = \mathsf{X}^{2j+1}\chi$.*
- *For $\psi = \psi_1 \,\mathsf{U}\, \psi_2$, we set $f'^{\mathrm{LTL}}(\psi) = ((\chi \wedge \mathsf{X}\chi \wedge \mathsf{X}^2\neg\chi) \rightarrow \psi_1) \,\mathsf{U}\, ((\chi \wedge \mathsf{X}\chi \wedge \mathsf{X}^2\neg\chi) \wedge \psi_2)$.*

Proof. We show the lemma by structural induction. More specifically, we show that for every $w \in (2^{\mathsf{AP}})^\omega$ and $i \in \mathbb{N}$, we have $w, i \models \psi$ if and only if $w', 2(n+1) \cdot i \models f'^{\mathrm{LTL}}(\psi)$ for $w' = f^{\mathrm{LTL}}(w)$.

- Case $\psi = \mathbf{true}$, $\psi = \mathbf{false}$: trivial
- Case $\psi = x_j$ for some $x_j \in \mathsf{AP}$. The definition of f^{LTL} ensures that for all $j \in \{1, \ldots, n\}$, we have $x_j \in w_i$ if and only $\chi \in w'_{2i(n+1)+2j+1}$. Thus, we have $w', 2i(n+1) \models \mathsf{X}^{2j+1}\chi$ if and only if $w, i \models x_j$.
- Case $\psi = \psi_1 \wedge \psi_2$, $\psi = \psi_1 \vee \psi_2$, and $\psi = \neg\psi_1$: trivial
- Case $\psi = \psi_1 \,\mathsf{U}\, \psi_2$. Here, $\chi \wedge \mathsf{X}\chi \wedge \mathsf{X}^2\neg\chi$ is true precisely at time points $2(n+1)i$ for some $i \in \mathbb{N}$, thus anything of interest for evaluating $f'^{\mathrm{LTL}}(\psi)$ happens at these time instants. For $f'^{\mathrm{LTL}}(\psi)$ to be true, we must have that for some i, w', we get that $2(n+1)i \models f'^{\mathrm{LTL}}(\psi_2)$, which by the induction hypothesis is equivalent to $w, i \models \psi_2$. For all $j < i$, we must have $w', 2(n+1)j \models f'^{\mathrm{LTL}}(\psi_1)$, which by the induction hypothesis is equivalent to $w, j \models \psi_1$.

□

Reconsider the setting from Figure 8.4. As an example, the specification $x_4 \,\mathsf{U}\, x_3$ for the original case translates to $((\chi \wedge \mathsf{X}\chi \wedge \mathsf{X}^2\neg\chi) \rightarrow \mathsf{X}^9\chi) \,\mathsf{U}\, ((\chi \wedge \mathsf{X}\chi \wedge \mathsf{X}^2\neg\chi) \wedge \mathsf{X}^7\chi)$ for the compressed case. In both variants, the specification is not fulfilled for this example.

Proving a claim as the one in Lemma 3 for safety automata instead of LTL is actually simpler than the case above, as we do not need to blow up a word by a factor of $2(n+1)$, as a trivial encoding with a blow-up of n suffices.

Using Lemma 3, we can now show that internal signals in a symmetric architecture easily lead to undecidability.

Theorem 7. *Synthesis from LTL or safety word automata specifications for the S0 architecture depicted in Figure 8.3 is undecidable.*

Proof. We prove this fact by first deriving the undecidability of a more complex architecture, and then apply word compression to show the claim for the S0 architecture.

Consider the architecture S2 in Figure 8.5. Every process has three inputs signals and three output signals. Let ψ be a specification for the A0 architecture, which was shown to be undecidable earlier. We show that there exists an implementation for the specification ψ in the A0 architecture if and only if there exists a joint implementation for the two processes in the S2 architecture that satisfies $\psi' = \psi_d \wedge \mathsf{G}(v \leftrightarrow \mathsf{X}o) \wedge \mathsf{G}(w \leftrightarrow \mathsf{X}p)$, where ψ_d results from prefixing all occurrences of the signals y and z in ψ with an LTL next-time operator. Without loss of generality, we assume that ψ encodes the termination of a Turing machine and is structured in the way as described in Chapter 6.4.

$\Rightarrow$: If we have an implementation for the A0 architecture satisfying ψ, then the implementation needs to be a symmetric one: both processes output the same bitstream when reading a 1 from their local

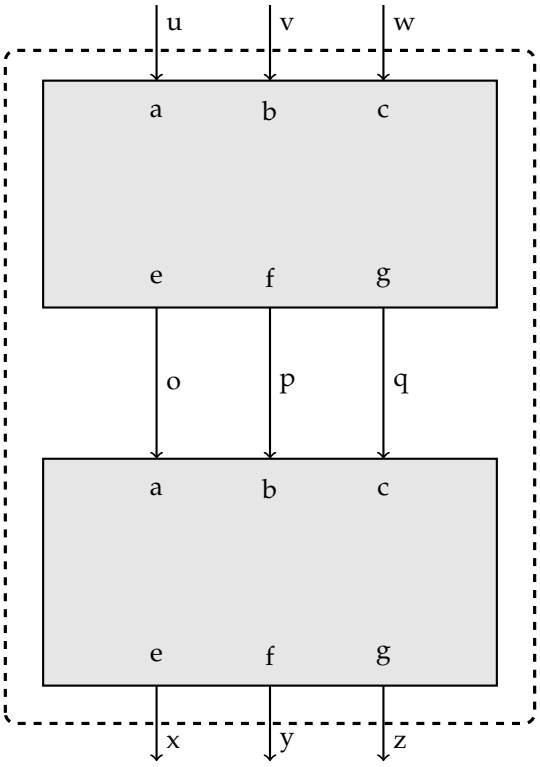

Figure 8.5: The undecidable *S2* symmetric architecture

input signal for the first time. We can take an implementation for one of these processes, and turn it into an implementation for a process in the S2 architecture: just simulate the process over the input signal a and use g as the local output for the tape content. At the same time, copy all values from b to e, and c to f. This makes sure that $\mathsf{G}(v \leftrightarrow \mathsf{X}o) \wedge \mathsf{G}(w \leftrightarrow \mathsf{X}p)$ is satisfied by the resulting system. Since the bottom process in the S2 architecture then outputs q with one computation cycle of delay to y, y in the S2 architecture always represents the output of the left process in the A0 architecture with a delay of one cycle. For the signal line from v to z, the same line of reasoning holds: the data from v appears at the signal o with a delay of one, and then, since every process in the S2 architecture simulates a process in the A0 architecture reading from a and writing to g, z is the output of the A0 process for the input v with a delay of one cycle. Thus, ψ_d is fulfilled by the system. Taking all of these facts together, the architecture with the implementation also fulfils ψ.

$\Leftarrow$: Assume that ψ' is realisable for the S2 architecture and we have a process that ensures that ψ' is satisfied by the resulting aggregated Moore machine. We argue that the process has to behave like a process for the A0 architecture for the input a and output g. The important feature of this architecture is that the process does not know if the local input b is the (delayed) a input to the other process, or if its c input is the (Turing machine tape) output of the other process. Thus, it cannot find out if it is the top process or the bottom process in the architecture and must prevent violating the specification in either case.

Recall that without loss of generality, we assumed ψ_D to encode the computation/acceptance of a Turing machine. First of all, in order to satisfy the specification, both processes have to output the first two Turing tape computations on their g outputs when obtaining a 1 to the a input. This follows from the fact that the specification only allows the processes to forward information from the inputs b and c, so the overall requirement to start with the first two tape contents on y when reading a true-bit on u, and to start with the first two tape contents on z when reading from v can only be fulfilled by distributing the writing of the two tapes among the processes.

Now assume that a process receives the first Turing tape configurations on local input c, then a start signal on input a while the second Turing tape content starts, and after both Turing tape contents have been seen, we get a starting signal on b. Let this input stream be called the *reference stream*.

Since the process does not know whether it is the top-most one in the architecture, it also has to output

the third Turing tape configuration on its local output g after the first two of these, as the input to b might have been forwarded to the bottom-most process, where it triggered the other process to output the first two tape configurations. Then, ψ_D would be violated if the top-most process did not output the first three configurations. But then, if the process is the lower-most one, as the local input c might actually be the output of the top-most process, must also output the first three Turing tape configurations in order not to violate the specification. Note that the lower-most process must do that regardless of its local input b, as it is just forwarded garbage from global input w then.

But then, this means that the top-most process also has to output the first four Turing tape configurations when reading the reference stream by the same reasoning - it might be the top-most process, and since it has forwarded a signal that would trigger the other process to start the Turing tape computation one tape content later, it would otherwise violate the specification.

We can iterate this argument ad infinitum. Now if the process ensures that the overall system, when the process is instantiated in the S2 architecture, satisfies the specification, then this means that the Turing machine has to terminate — since we can force the system to output the correct Turing tape computations along a run of the Turing machine, and we also require it to be able to halt, there is no other way that the specification can be satisfied.

Since we can compress (1) the input signals u,v, and z into one signal (named x in the S0 architecture), (2) o, p, and q into one signal, (3) u, v, and z into one signal, and (4) adapt ψ' accordingly (and all of these compressions can use the same encoding), the undecidability of the S0 architecture follows. Definition 28 and Lemma 3 describe how this word compression step can be performed. However, we need to make sure that the correct functioning of the processes in the S0 architecture is only enforced on input streams that result from compressing a word over $2^{u,v,w}$ according to Definition 28. For this, we take the adapted version of ψ' and replace it by $\psi' \vee \mathsf{F}\phi_{invalid1} \vee \phi_{invalid2}$, where $\phi_{invalid1}$ and $\phi_{invalid2}$ encode that a part of the input stream is found that shows that it can not have been obtained by word compression according to Definition 28. Formally, we can describe these properties as:

$$\begin{aligned}\phi_{invalid1} &= x \wedge \mathsf{X}x \wedge \left(\neg\mathsf{X}^8 x \vee \neg\mathsf{X}^9 x \vee \bigvee_{i\in\{1,\dots,3\}} \mathsf{X}^{2i} x\right)\\ \phi_{invalid2} &= \neg x \vee \neg\mathsf{X}x\end{aligned}$$

Intuitively, $\phi_{invalid1}$ states that starting from a character start sequence, the next character start sequence does not come after exactly 8 cycles (or we have an illegal bit encoding along the way), and $\phi_{invalid2}$ states that the input stream does not start with a character start sequence.

Note that if ψ is a safety specification, then we can also write the final specification as a safety automaton by the small modification that instead of using $\psi' \vee \mathsf{F}\phi_{invalid1} \vee \phi_{invalid2}$, we take $\psi' \vee \phi_{invalid2}$, translate it to a safety word automaton, and modify it such that the automaton enters an accepting absorbing state once a part of the word satisfying $\phi_{invalid1}$ has been witnessed. In this way, the system may only behave arbitrarily *after* the input stream has become invalid, but this does not change the realisability of the specification. Thus, the claim also follows for specifications written in form of safety automata and not only LTL. □

8.2.2 No internal signals – Undecidability

In Chapter 8.2.1, we have seen that feeding the output of one process as an input to another process in a symmetric architecture can easily lead to undecidability, even in the case that every process only has one input and output signal each. Let us now analyse the case that we do not have *cascaded* processes, i.e., no process can read the output of another process. Note that in the A0 architecture undecidability proof, for the specification that we build, there can only exist one implementation (modulo isomorphism and quotienting) for each of the processes such that the overall system satisfies the specification, and the two implementations are actually identical. We use this fact to derive the undecidability of a greater class of symmetric architectures here. Similarly to the work by Finkbeiner and Schewe (2005), where the *information fork* criterion for undecidable non-symmetric architectures is established, we search for a pair of incomparably informed processes in a symmetric architecture to prove its undecidability.

Proposition 1. *Let $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ be a symmetric architecture over the interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$. If there are two tuples (p, x) and (p', x') in $P \times \mathsf{AP}^I_g$ such that*

- $p \neq p'$,
- $\exists y \in \mathsf{AP}^I : E^{in}(p, y) = x$,
- $\exists y' \in \mathsf{AP}^I : E^{in}(p', y') = x'$,
- $\nexists y \in \mathsf{AP}^I : E^{in}(p, y) = x'$, *and*
- $\nexists y' \in \mathsf{AP}^I : E^{in}(p', y') = x$,

and no output of one process is used as an input to another process, then the symmetric synthesis problem for $\mathcal{E}$ for LTL specifications is undecidable.

Proof. We prove the claim by reducing the realisability checking problem for a specification ψ for the A0 architecture onto a symmetric realisability checking problem for $\mathcal{E}$. Without loss of generality, we can assume that ψ only allows implementations in which both processes behave in the same way.

We take as modified specification $\psi' = (\mathsf{G} \bigwedge_{k \in AP_g^I \setminus \{x,x'\}} \neg k) \rightarrow \psi[x/u][x'/v][E^{out}(p,s)/y][E^{out}(p',s)/z]$ for some fixed but arbitrary $s \in \mathsf{AP}^O$. There exists a one-to-one correspondence between implementations satisfying ψ in the A0 architecture and implementations for ψ' in $\mathcal{E}$.

Starting with a process implementation for the A0 architecture, we can translate it to one for $\mathcal{E}$ and ψ' by letting the process in $\mathcal{E}$ take the disjunction of all input bits, and simulating the old process for the disjunction as input. The process then writes the output of the simulated A0 process to all local outputs (or, alternatively, to s). The way in which ψ' is composed guarantees that the input and output bits need not be distinguished by the process in the architecture $\mathcal{E}$.

Starting with a process implementation for $\mathcal{E}$ and ψ', we can translate it to an implementation for the A0 architecture as follows: we simulate the process for $\mathcal{E}$ by duplicating the local input of the process in the A0 architecture to the local inputs y and y' for $y, y' \in \mathsf{AP}^I$ such that $E^{in}(p, y) = x$ and $E^{in}(p', y') = x'$, and constantly feed 0 to all other inputs. The local output s of the process for $\mathcal{E}$ is then used as the local output for the process in the A0 architecture. □

Performing the same proof is a bit more difficult for specifications written as safety automata, even if ψ is a safety constraint, as the formula $\psi' = (\mathsf{G} \bigwedge_{k \in AP_g^I \setminus \{x,x'\}} \neg k) \rightarrow \psi[x/u][x'/v][E^{out}(p,s)/y][E^{out}(p',s)/z]$ as used in the proof of Proposition 1 is actually not a safety formula, and thus has no equivalent safety automaton. However, we can fix this by applying the same basic idea as in the proof of Theorem 7. More precisely, we can build a *weak automaton* from the specification, with one non-accepting level for the case that $\psi[x/u][x'/v][E^{out}(p,s)/y][E^{out}(p',s)/z]$ has already been violated, but not yet $(\mathsf{G} \bigwedge_{k \in AP_g^I \setminus \{x,x'\}} \neg k)$. Note that in such a case, the environment can enforce that ψ' is not satisfied by feeding **false** to all input signals henceforth. Thus, for the scope of synthesis, we can just merge the non-accepting level of states into $\bot$ and obtain an equi-realisable safety specification.

8.2.3 No internal signals – Decidability

After all of these negative results, let us now identify a large class of symmetric architectures whose synthesis problems we prove to be decidable in this thesis. The main property of this class is that all processes obtain all input to the overall system, but the input is rotated depending on their process identifiers. This requirement prevents the existence of information forks. At the same time, all processes have the same knowledge about how much *symmetry has been broken* by the input already. We start by recalling some basic definitions from group theory, and then define *rotation-symmetric architectures*. We illustrate the ideas by examples and describe fundamental properties of these architectures.

Definition 29. *A* finite cyclic group *of order n is a group with elements $\{p^0, p^1, p^1, \ldots, p^{n-1}\}$, i.e., it is generated by some element p, and $p^n = p^0$ is the neutral element of the group.*

Definition 30. *Given two groups G^1 and G^2 with element sets X^1 and X^2 and the* group operations $\cdot_1$ *and* $\cdot_2$*, we say that some group G is the* direct group product *of G^1 and G^2 if it has the set of elements $X^1 \times X^2$ and its group operation is defined as $(x^1, x^2) \cdot (x'^1, x'^2) = (x^1 \cdot^1 x'^1, x^2 \cdot^2 x'^2)$.*

Definition 31. *Given some set X, we call a group whose elements are permutations of X a* permutation group *if the group operation permutes one operand by the permutation defined by the other operand.*

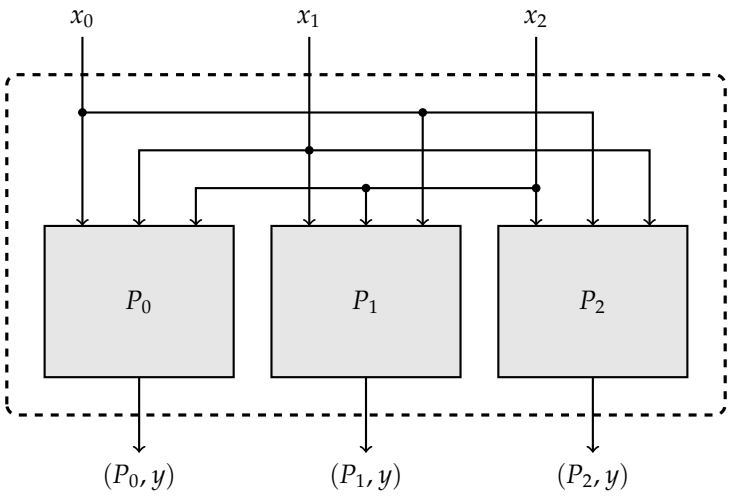

Figure 8.6: A simple rotation-symmetric architecture.

An example for a permutation group is the one with the elements $\{(1,2,3,4),(1,2,4,3),(1,3,2,4),(1,3,4,2),(1,4,2,3),(1,4,3,2)\}$, and computing $(1,3,2,4)\cdot(1,4,3,2)$ would result in $(1,4,2,3)$.

Let us now introduce a new technical term for a special type of groups to simplify the presentation in the following.

Definition 32. *Let G be a cyclic group of order n with generating element p. We call the permutation group G' with the same number of elements* corresponding *to G if G' is generated by the element $(p^1, p^2, \ldots, p^{n-1}, p^0)$.*

Definition 33. *Given a set X, we call a permutation group P with the set of elements X* rotation-symmetric *if and only if it is a direct group product of a set of permutation groups that correspond to cyclic groups.*

Definition 34. *A symmetric architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ over the interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ with n processes is called* rotation-symmetric *if and only if there exists some rotation-symmetric permutation group G with n elements and following conditions hold:*

- *w.l.o.g., $\mathsf{AP}^I = \mathsf{AP}^I_L \times \{0, \ldots, n-1\}$ and P is the set of elements of G,*
- $\mathsf{AP}^I = \mathsf{AP}^I_g$,
- *S is the product of AP^O and the elements of G,*
- *for every $Y = (y_0, \ldots, y_{n-1})$ in G, $x \in \mathsf{AP}^I_L$, and $i \in \{0, \ldots, n-1\}$, we have $E^{in}(Y, (x, i)) = y_i$, and*
- *for every $Y = (y_0, \ldots, y_{n-1})$ in G and $x \in \mathsf{AP}^O$, we have $E^{out}(Y, x) = (Y, x)$.*

To illustrate this definition, let us discuss an example. Figure 8.6 describes a simple rotation-symmetric architecture. The architecture has three input signals and three output signals. The first process obtains the values of the global input signals in the order (x_0, x_1, x_2) along its local input signals, the second process obtains (x_1, x_2, x_0), and the third process gets (x_2, x_0, x_1). In a sense, the input order is rotated for every process, which justifies the term *rotation-symmetric* for such symmetric architectures. Formally, we have $\mathsf{AP}^I = \{x_0, x_1, x_2\}$ and $\mathsf{AP}^O = \{y\}$ here.

Our definition is however not bound to systems in which the processes form only one *cycle* (i.e., have a cyclic group as their underlying permutation group). Figure 8.7 shows a more complex example in which the permutation group is a product of the permutation groups induced by two cyclic groups of order two. For a process $P_{i,j}$, we have that i is 1 if and only if x_0 and x_1 are swapped for the local input of a process, and j is 1 if and only if x_2 and x_3 are swapped. For example, process $P_{1,0}$ reads the inputs in the order (x_1, x_0, x_2, x_3).

All rotation-symmetric architectures have the property that all processes get all input signals, thus avoiding information forks. We will see in the next chapter that having a rotation-symmetric architecture is also a sufficient criterion for the decidability of the symmetric synthesis problem. Thus, we will obtain the following result (accompanied with a synthesis procedure and a complexity analysis):

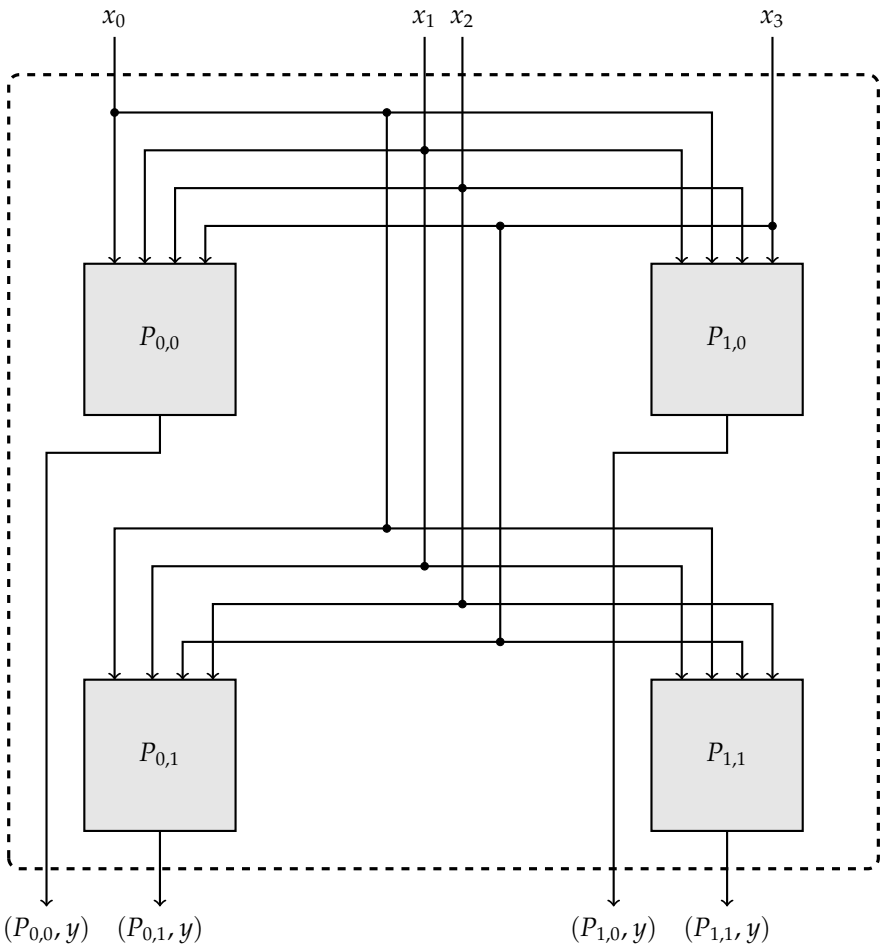

Figure 8.7: A more complex rotation-symmetric architecture.

Theorem 8. *The symmetric synthesis problem for rotation-symmetric architectures is decidable for specifications in LTL, CTL, or as ω-regular word automata of any acceptance condition and branching type discussed in this thesis.*

Rotation-symmetric architectures are a natural model for symmetric systems in which all processes obtain the global input. Often, a simple ring structure is given by the spatial structure of the application. For example, in a traffic light system for a street junction, the four traffic lights are naturally aligned along a *cycle* of order four, as depicted in Figure 8.8. Likewise, rotation sorters, which are devices to forward packets or other goods in logistics applications, have a spatial cyclic structure. Such a system is depicted in Figure 8.9. The burden to forward all input to all components is often not a significant one in practice, as the bandwidth required is typically low. On the other hand, as we obtain an automated synthesis procedure for such settings in the next chapter, the benefit of doing so can be huge. Sometimes, it is reasonable to assume that some inputs arrive later than others, especially in a ring communication structure. We will show how to deal with such situations in Section 9.3, thus lifting the usefulness of the synthesis procedure outlined next to such settings.

While the requirement for the group to be rotation-symmetric in Theorem 8 might seem restrictive, in fact it is not. Frobenius and Stickelberger (1878) showed that every finite group can be written as a direct group product of a finite number of cyclic groups, each having a number of elements that is a power of a prime. As cyclic groups are isomorphic to their respective permutation group (using Definition 33), our rotation-symmetry definition for architectures is actually quite general.

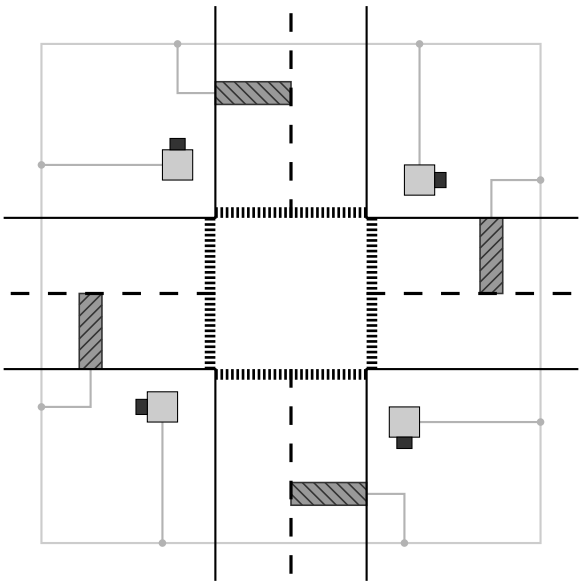

Figure 8.8: A distributed traffic light controller as a rotation-symmetric system that consists of a single cycle of length 4. For every direction of the crossing, a sensor detects incoming cars, and there is a traffic light for signalling whether cars from that direction may enter the crossing. All input is forwarded along a circular data bus to all lights, where the controllers can make decisions locally. In case of a defect of one controller, the overall system can still operate, provided that the defect is signalled as a "yield to everyone" command to drivers coming from the direction to which the damaged controller corresponds.

8.3 Concluding remarks

Before showing how to solve the symmetric synthesis problem for rotation-symmetric architectures in the next chapter, let us quickly recapitulate the content of this chapter. We have introduced symmetric architectures and discussed that feeding one process' output to another process as an input can easily lead to undecidability of the synthesis problem for the architecture. Then, for the remaining cases, we discussed that architectures with incomparably informed processes are also typically undecidable. Finally, we have characterised a large class of systems whose synthesis problem we will show to be decidable in the next chapter.

We did not perform a thorough partitioning of the architectures into decidable ones and undecidable ones, but concentrated on identifying the driving factors for undecidability and characterised a huge class of symmetric architectures that is likewise useful and highly non-trivial.

Among the architectures that we have not identified as being decidable or undecidable above, there are many special cases that require specialised arguments to analyse them. Take for example the architecture in Figure 8.10, which is not covered by the cases discussed in the previous sections. Both processes receive the same input in the same order and thus, for a symmetric implementation, must produce the same output. Thus, the synthesis problem for this architecture can be solved by taking the conjunction of the original specification with $\mathsf{G}((P_0, y) \leftrightarrow (P_1, y))$ (for LTL specifications – in the CTL or word automaton case, however, a similar operation can be performed) and applying a normal synthesis procedure for non-distributed ("*monolithic*") systems. In case of a positive result, we just chop away one output signal to obtain an implementation for one process. For some slightly more complicated architectures, such as the one in Figure 8.11, we need to alternate the idea a bit: the processes have to output the same values until they can distinguish which process number they have. Thus, we would apply a monolithic procedure for the specification $\psi \wedge ((\neg(a \leftrightarrow b))\mathsf{R}((P_0, y) \leftrightarrow (P_1, y)))$ if ψ is the original specification for the overall system. If and only if the modified specification is realisable in the monolithic case, there exists a symmetric implementation for the setting, and we can easily obtain a process' implementation by letting the process simulate the monolithic implementation until $a \neq b$, picking any of the output signals of the monolithic implementation as its own output. As at the point

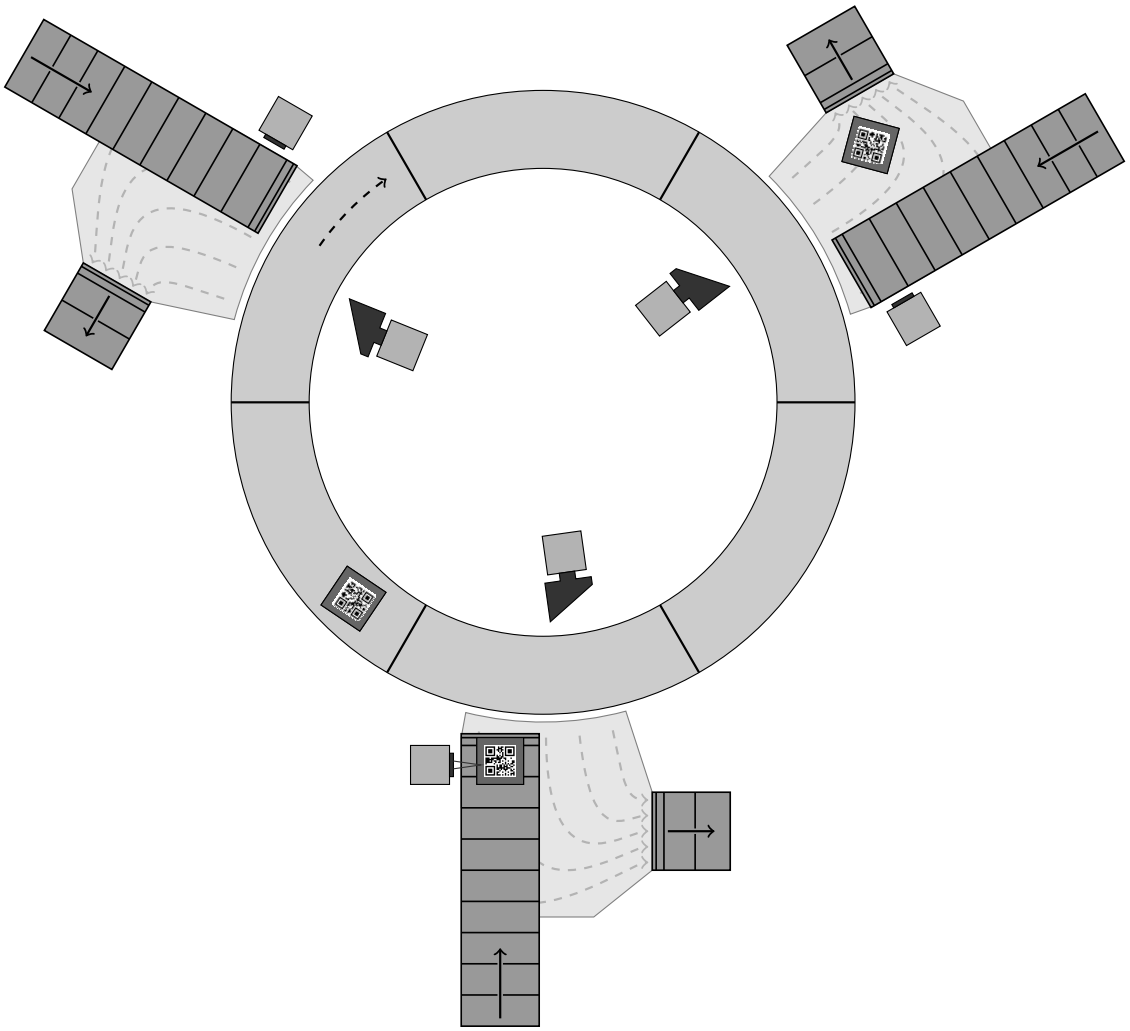

Figure 8.9: A rotation sorter. Packets arrive on conveyor belts from three directions. For each belt for incoming packets (the "in-belts"), there is a barcode reader, which the system can use to read the destination of the packet. We assume that the in-belts are controlled, so that the sorter can decide when to push a packet onto the platform. In the centre, there are some arms that can push the packet off the platform when it has reached its destination. A slide then transports the packet to the respective outgoing conveyor belt. A distributed controller for the in-belts and the arms is implementable in a rotation-symmetric fashion if the barcode reader provides the controller with the relative destination addresses of the packets (i.e., if the packet is to be pushed from the platform after transporting it by 120, 240, or 360 degrees).

in which $a \neq b$ holds, every process knows its identity (by checking whether the middle input is equal to the left-most one or the right-most one), the processes can continue to simulate the monolithic implementation along their local outputs then.

Partitioning the symmetric architectures into the ones with decidable and undecidable synthesis problems is closely related to the question of *distributed implementability*. The applications under concern in that field of research have different properties and the proofs for non-implementability as distributed systems only apply to a few applications each. As Fich and Ruppert (2003) note, "A comprehensive survey of impossibility results in distributed computing would require an entire book". The setting here is similar in the sense that different architectures under concern lend themselves to different arguments for the undecidability or decidability of their respective symmetric implementation problems. Nevertheless, the cases dealt with above cover the majority of architectures found in practice, and closing the remaining special cases is left as future work.

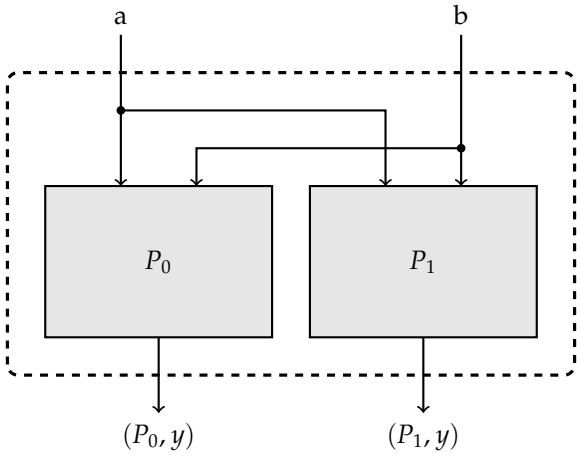

Figure 8.10: A decidable symmetric architecture that is not rotation-symmetric.

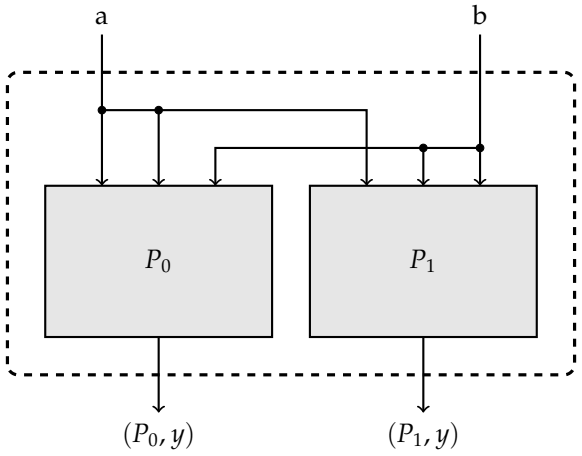

Figure 8.11: A more complex decidable symmetric architecture that is not rotation-symmetric.

Chapter 9

Algorithms for symmetric synthesis

In the previous chapter, we identified rotation-symmetric architectures as a sub-class of the symmetric architectures for which fully automatic synthesis is highly desirable. In this chapter, we prove the synthesis problem for this architecture class to be decidable for specifications in linear-time temporal logic and discuss a novel synthesis procedure whose complexity matches the complexity of the problem. Later, we also discuss the problem for specifications in the branching-time logic CTL* or as ω-word automata, and provide synthesis procedures for these cases. To simplify the presentation, we will only deal here with architectures for which the underlying rotation-symmetric permutation groups consist of a single cycle each, as we can save a lot of notation in this way. The extension of the concepts to the general case is however described in Section 9.2.

The key insight that allows us to solve the symmetric synthesis problem for rotation-symmetric architectures is that we can *decompose the symmetry requirement into two sub-requirements*:

- Symmetry shall not be broken by the distributed implementation ahead of time.
- The behaviour of the system is *rotation-invariant*.

The first requirement is ω-regular and can be captured by an LTL formula or a safety word automaton. The second property is however not, and we will apply an automaton modification procedure to account for this requirement.

Decomposing a non-regular property in into a regular one and another non-regular one that we can nevertheless deal with on the automaton level is a technique that is, to the best of the author's knowledge, not used anywhere else in the synthesis literature before. It is fair to conjecture that the applicability of this decomposition concept extends far beyond the synthesis of rotation-symmetric systems.

In the next section, we start the presentation by introducing some simplifying notation for the concepts to follow and then motivate and describe the two properties into which we decompose the symmetry requirement for the synthesised solution. We prove that every rotation-symmetric system has these properties and that together they capture the set of rotation-symmetric implementations. Then, we show how to use this insight in a synthesis procedure for LTL specifications. In Section 9.1.5, we prove that this construction is optimal in the complexity-theoretic sense. Finally, in Section 9.1.6, we extend it to specifications in CTL*. The resulting synthesis procedure can in fact deal with arbitrary alternating Rabin tree automaton specifications, and thus can also be used for ω-word automaton specifications by spreading these to such tree automata.

9.1 Properties of rotation-symmetric systems

A key idea in the following synthesis algorithm for rotation-symmetric systems is to synthesise a computation tree of the aggregated Moore machine of the processes rather than a computation tree for a single process. Let us have another look at how computation trees induced by aggregated Moore machines look like. For simplicity, we first give a more direct definition of the aggregated Moore machine of a rotation-symmetric system, the so-called *symmetric product* of a Moore machine. Then, we define a *symmetry property* for computation trees and prove that it captures precisely the computation trees induced by symmetric products.

To simplify the presentation in the following, we assume that the processes are called $P = \{P_0 \dots P_{n-1}\}$ for some value $n \in \mathbb{N}$, and that we have fixed some rotation-symmetric architecture $\mathcal{E} = (S, P, \mathsf{AP}^I_g, E^{in}, E^{out})$ over some local process interface $(\mathsf{AP}^I_L, \mathsf{AP}^O_L)$ with a single cycle of processes. We define $\mathcal{I} = 2^{\mathsf{AP}^I_g}$ to denote the global input alphabet to all processes, and $\mathcal{O} = 2^{\{\mathsf{AP}^O_L \times P\}}$ is the global output. The local output of one process on the other hand is given as $O = 2^{\mathsf{AP}^O_L}$. To compress notation a bit and enhance

the readability of the expressions in the following, for an atomic proposition in some set $\mathsf{AP} \times P$, we attach the process identifier as a subscript to an element $x \in \mathsf{AP}$, and write x_i instead of (x, i) for $i \in P$.

The following rotation function will become useful in the analysis below. Let U be some set such that $U = 2^{V \times P}$ for some other set V. We define a *rotation operator* over U by setting $\mathrm{rot} : U \times \mathbb{Z} \to U$ with $\mathrm{rot}(X, k) = \{(x, (j + k) \bmod n) \mid (x, j) \in X\}$ for every $X \in U$ and $k \in \mathbb{Z}$. Furthermore, we extend rot to LTL formulas and define $\mathrm{rot}(\psi, k)$ for an LTL formula ψ over the set of variables $\mathsf{AP} \times P$ and $k \in \mathbb{Z}$ to be ψ with all atomic propositions (p, j) replaced by $(p, (j + k) \bmod n)$ for $p \in \mathsf{AP}$, $j \in \mathbb{Z}$. For clarity, when dealing with the rot function for some set $Z = 2^{V \times P}$, we often partition the elements of $V \times P$ by their process indices and for example write $(X_0, \ldots, X_{n-1})$ instead of $(X_0 \times \{0\}) \cup \ldots \cup (X_{n-1} \times \{n-1\})$ for $X_0, \ldots, X_{n-1} \subseteq \mathsf{AP}$. In case V is a singular set (which will often be the case in the examples to follow), we will also simply use 0 and 1 to represent whether the element in V is in x_j for some $j \in \mathbb{N}$ or not. The rotation function is extended to sequences of elements in U by rotating the individual sequence items.

Definition 35 (Symmetric product)**.** *Given a Moore machine $\mathcal{M} = (Q, \mathcal{I}, O, \delta, q_0, L)$, we say that a Moore machine $\mathcal{M}' = (Q', \mathcal{I}, \mathcal{O}, \delta', q_0', L')$ is the symmetric product of $\mathcal{M}$ if $Q' = Q^n$, $q_0' = (q_0)^n$, and for all $q_0'', \ldots, q_{n-1}'' \in Q$, $(i_0, \ldots, i_{n-1}) \in \mathcal{I}$:*

$$\delta'((q_0'', \ldots, q_{n-1}''), (i_0, \ldots, i_{n-1})) = (q_0''', \ldots, q_{n-1}''')$$

s. t. $\forall 0 \leqslant j < n : q_j''' = \delta(q_j'', \mathrm{rot}((i_0, \ldots, i_{n-1}), -j))$ and $L'(q_0'', \ldots, q_{n-1}'') = (L(q_0''), L(q_1''), \ldots, L(q_{n-1}''))$.

Note that Definition 35 is just a combination of Definition 24 and Definition 34, applied to architectures consisting of a single cycle of processes.

Definition 36 (Symmetry property)**.** *Given a tree $\langle T, \tau \rangle$ over $T = \mathcal{I}^*$ and $\tau : T \to \mathcal{O}$, we say that the tree has the* symmetry property *if for each $t \in T$ and $0 \leqslant i < n$, $\tau(\mathrm{rot}(t, i)) = \mathrm{rot}(\tau(t), i)$.*

We now establish the fact that the symmetry property captures precisely the computation trees induced by some symmetric product.

Lemma 4 (Symmetry lemma)**.** *The set of regular trees having the symmetry property is precisely the same as the set of trees that are induced by the symmetric product of some Moore machine.*

Proof. $\Leftarrow$: The fact that the computation tree induced by the symmetric product of some Moore machine has the symmetry property follows directly from the definitions.

$\Rightarrow$: For the converse direction, we prove that from every regular computation tree with the symmetry property, we can construct a Moore machine that is an implementation for one process, and by taking the symmetric product of the Moore machine, we obtain a product machine whose computation tree is in turn the one that we started with.

Let $\langle T, \tau \rangle$ be the computation tree to start with. As it is regular, we have an equivalence relation over the nodes in the tree. Let $[\cdot]$ be the function that maps a tree node in t onto a tree node representing its equivalence class, so for all $t, t' \in T$, we have that the sub-trees induced by t and t' are the same if and only if $[t] = [t']$, and for every t there is some t' such that $[t] = t'$. We build a Moore machine for one process in the symmetric architecture from $\langle T, \tau \rangle$ by setting $\mathcal{M} = (S, \mathcal{I}, O, \delta, s_{init}, L)$ with:

$$\begin{aligned} S &= \{[t] \mid t \in T\} \\ \delta(s, x) &= [sx] \text{ for all } x \in \mathcal{I} \text{ and } s \in S \\ s_{init} &= [\epsilon] \\ L(s) &= \tau(s)|_{O_0} \text{ for all } s \in S \end{aligned}$$

We now show that the symmetric product of $\mathcal{M}$ induces a computation tree that is the same as $\langle T, \tau \rangle$. If we take the symmetric product (Definition 35) of $\mathcal{M}$, we obtain $\mathcal{M}' = (S', \mathcal{I}, \mathcal{O}, \delta', s_{init}', L')$ with:

$$\begin{aligned} S' &= \{([t_0], \ldots, [t_{n-1}]) \mid t_0, \ldots, t_{n-1} \in T\} \\ \delta'((t_0, \ldots, t_{n-1}), x) &= ([t_0 \,\mathrm{rot}(x, 0)], [t_1 \,\mathrm{rot}(x, -1)], \ldots, [t_{n-1} \,\mathrm{rot}(x, -n+1)]) \\ s_{init} &= ([\epsilon], \ldots, [\epsilon]) \\ L'((t_0, \ldots, t_{n-1})) &= (\tau(t_0)|_{O_0}, \tau(t_1)|_{O_0}, \ldots, \tau(t_{n-1})|_{O_0}) \end{aligned}$$

Let $\langle T', \tau' \rangle$ be the extended computation tree induced by $\mathcal{M}'$ with $\tau' : T' \to S' \times O$. We can show by induction that for every $t \in T$, we have that $\tau'(t)|_{S'} = ([\mathrm{rot}(t,0)], [\mathrm{rot}(t,-1)], [\mathrm{rot}(t,-2)], \ldots, [\mathrm{rot}(t,-n+1)])$. The induction basis is trivial, as $\tau'(t)|_{S'}(\epsilon) = ([\epsilon], [\epsilon], [\epsilon], \ldots, [\epsilon])$. For the inductive step, we have:

$$
\begin{aligned}
& \tau'(tx)|_{S'} && (9.1)\\
= \; & ([t_0\,\mathrm{rot}(x,0)], [t_1\,\mathrm{rot}(x,-1)], \ldots, [t_{n-1}\,\mathrm{rot}(x,-n+1)]) && (9.2)\\
& \text{for } \tau'(t)|_{S'}(t) = (t_0, t_1, \ldots, t_{n-1}) \\
= \; & ([[\mathrm{rot}(t,0)]\,\mathrm{rot}(x,0)], \ldots, [[\mathrm{rot}(t,-n+1)]\,\mathrm{rot}(x,-n+1)]) && (9.3)\\
= \; & ([\mathrm{rot}(t,0)\,\mathrm{rot}(x,0)], \ldots, [\mathrm{rot}(t,-n+1)\,\mathrm{rot}(x,-n+1)]) && (9.4)\\
= \; & ([\mathrm{rot}(tx,0)], \ldots, [\mathrm{rot}(tx,-n+1)]) && (9.5)
\end{aligned}
$$

In step (9.1)-(9.2) of this deduction, we applied the definitions of the elements of $\mathcal{M}$ and $\mathcal{M}'$. In step (9.2)-(9.3), we used the inductive hypothesis. In step (9.3)-(9.4), we used the regularity of the tree: for some $t \in T$ and $x \in \mathcal{I}$, we need to have $[[t]x] = [tx]$ as the subtree induced by $[t]x$ has to be the same as the one induced by tx, as otherwise $[t]$ and t would not be in the same equivalence class of subtrees (which is a contradiction). The last step uses the fact that if we concatenate two strings that are rotated by the same number of indices, then we can also first concatenate, and then rotate.

Now let us have a look at the outputs in the extended computation tree $\langle T', \tau' \rangle$. For every $t \in T'$, we have:

$$
\begin{aligned}
& \tau'(t)|_O && (9.6)\\
= \; & L'(\tau'(t)|_{S'}) && (9.7)\\
= \; & L'([\mathrm{rot}(t,0)], \ldots, [\mathrm{rot}(t,-n+1)]) && (9.8)\\
= \; & (\tau([\mathrm{rot}(t,0)])|_{O_0}, \ldots, \tau([\mathrm{rot}(t,-n-1)])|_{O_0}) && (9.9)\\
= \; & (\tau(\mathrm{rot}(t,0))|_{O_0}, \ldots, \tau(\mathrm{rot}(t,-n-1))|_{O_0}) && (9.10)\\
= \; & (\tau(t)|_{O_0}, \tau(t)|_{O_1}, \ldots, \tau(t)|_{O_{n-1}}) && (9.11)\\
= \; & \tau(t) && (9.12)
\end{aligned}
$$

In step (9.8)-(9.9), we simply applied the definition of L'. In step (9.9)-(9.10), we used the fact that we are dealing with equivalence classes over nodes in the computation tree $\langle T, \tau \rangle$ that respect the labelling of the system. In step (9.10)-(9.11), we use the symmetry property of $\langle T, \tau \rangle$. For every $i \in \{0, \ldots, n-1\}$, we have $\tau(\mathrm{rot}(t,i))|_{O_0} = \mathrm{rot}(\tau(t), i)|_{O_0}$ by this property, and then $rot(\tau(t), i)|_{O_0} = \tau(t)|_{O_i}$ by renaming. In the last step, we just plug together the tuple.

□

Now that we have identified the basic properties of computation trees induced by rotation-symmetric systems, let us move our focus back towards the synthesis problem. The following theorem shows that symmetry of a computation tree is not a regular property.

Theorem 9 (Finkbeiner, 2010)**.** *The set of symmetric computation trees for the rotation-symmetric architecture $\mathcal{E}$ with interface $\mathcal{I} = (\mathsf{AP}_L^I \times \{0,1\}, \mathsf{AP}_L^O)$ for the process and $\mathsf{AP}_L^I = \{i\}$ and $\mathsf{AP}_L^O = \{o\}$ is not a regular tree language.*

Proof. For a proof by contradiction, suppose that the set of symmetric computation trees is regular. The language includes a tree with the symmetry property in which the node labels on the path $(\varnothing, \{i\})^*$ and, symmetrically, on the path $(\{i\}, \varnothing)^*$ form the sequence $l = (\varnothing, \varnothing)^1(\{o\}, \{o\})(\varnothing, \varnothing)^2(\{o\}, \{o\}) \ldots$, i.e., the length of the $(\varnothing, \varnothing)$-sequences grows according to the distance to the root. According to the pumping lemma for regular tree languages, however, the sequence l can be partitioned into $l = u \cdot v \cdot w$, such that, for every $k > 0$, there exists a tree in the language where the label sequence on $(\varnothing, \{i\})^*$ is $l = u \cdot v^k \cdot w$, while the label sequence on $(\{i\}, \varnothing)^*$ is still l. Clearly, these trees are not symmetric. □

The non-regularity of symmetry tells us that incorporating a requirement to only output trees with the symmetry property into a synthesis procedure is non-trivial, as we cannot describe this requirement as a tree automaton. As a remedy, we decompose the symmetry requirement into two sub-requirements that we can enforce in a synthesis process. We next describe two necessary properties of symmetric computation trees, and later show that they are also sufficient in conjunction. This decomposition

Round		0	1	2	3	4	5	6	...
Output of the processes	P_0	0	0	1	0	0	0	0	...
	P_1	0	**1**	1	0	0	0	0	...
	P_2	0	0	1	0	0	0	0	...
	P_3	0	0	1	**1**	0	1	0	...
Input to process P_0		0	1	1	1	0	0	1	...
		0	1	0	1	0	0	0	...
		0	1	1	0	0	0	0	...
		0	1	0	1	0	0	1	...
Input to process P_1		0	1	0	1	0	0	0	...
		0	1	1	0	0	0	0	...
		0	1	0	1	0	0	1	...
		0	1	1	1	0	0	1	...
Input to process P_2		0	1	1	0	0	0	0	...
		0	1	0	1	0	0	1	...
		0	1	1	1	0	0	1	...
		0	1	0	1	0	0	0	...
Input to process P_3		0	1	0	1	0	0	1	...
		0	1	1	1	0	0	1	...
		0	1	0	1	0	0	0	...
		0	1	1	0	0	0	0	...

Table 9.1: Example for incorrect input/output symmetries for a rotation-symmetric architecture with four processes. The local input (i.e., the rotation of the global input that the individual processes see) is given for all four processes. The global input coincides with the local input to process P_0 and is thus not given in addition.

allows us to modify a classical synthesis procedure for monolithic systems to only produce symmetric computation trees, from which we can read off the process implementation in a rotation-symmetric system.

9.1.1 Symmetry breaking

A fundamental concept in the study of symmetric systems is *symmetry breaking*. When the overall systems starts, no process has information about its identity. Thus, initially, they all have to output the same values. The processes have to continue producing the same outputs until one process obtains a stimulus that allows it to distinguish itself from the other processes. In the traditional distributed systems literature, in which *leader election in networks* is a common topic (see, e.g., Fich and Ruppert, 2003; Itai and Rodeh, 1990), symmetry breaking is achieved by inter-process communication (which may take a non-deterministic amount of time) and the employment of random numbers. In the symmetric synthesis setting, where we want to avoid communication between processes as this makes implementing the system in the field more costly and typically leads to the undecidability of an architecture's synthesis problem, we do not have this possibility. Thus, the only way to break symmetry is by the input.

Let us discuss this idea by means of an example. Figure 9.1 describes a simple rotation-symmetric architecture with four processes, arranged as a ring. Recall that we assume a Moore-type computation model, so the output is produced first. In the first computation cycle, none of the processes have received any input so far. As they all have the same implementation, an aggregated Moore machine of a rotation-symmetric architecture can only output the same value on all output bits. Only when symmetry has been broken by input, the processes can output different values. In the example run of a system with four processes given in Table 9.1, in the first computation cycle, all processes produce the same output. In the second computation cycle, however, process P_1 outputs a different value than the others, although the input read so far has been the same. Thus, the run depicted cannot be part of the computation tree of a symmetric system.

Breaking the symmetry of a system is however no process that has either been done or not - it can also be broken *partially*. Reconsider the example run in Table 9.1. For the first two computation cycles, all

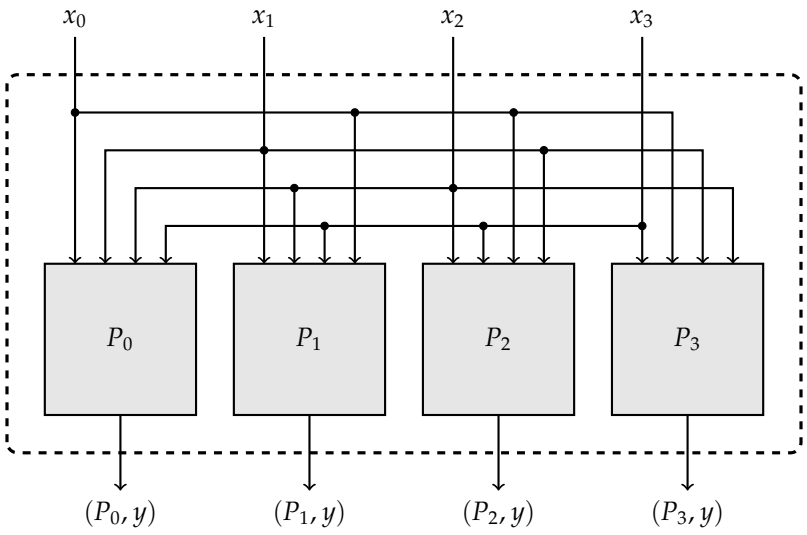

Figure 9.1: A single-cyle rotation-symmetric architecture with four processes.

processes receive the same input. In the third computation cycle, this situation changes: processes P_0 and P_2 receive a $(1,0,1,0)^T$, and processes P_1 and P_3 receive a $(0,1,0,1)^T$. After the first three computation cycles, the processes have thus received the following sequences as their local input streams:

$$P_0 : \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad P_1 : \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

$$P_2 : \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad P_3 : \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

As it can easily be seen, for processes P_0 and P_2, the input stream has been the same. The same holds for processes P_1 and P_3. Thus, for a rotation-symmetric implementation, processes P_0 and P_2 must output the same value in the fourth computation cycle, and so must P_1 and P_3. In Table 9.1, they actually do not do this. Thus, the forth computation cycle also witnesses that the system run depicted in Table 9.1 cannot stem from a rotation-symmetric system. Note, however, that the input in the fourth computation cycle breaks the symmetry completely. The local inputs to the four processes then have been as follows, and are all different:

$$P_0 : \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \qquad P_1 : \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

$$P_2 : \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad P_3 : \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Table 9.2 describes a run of a system that can be the outcome of executing a rotation-symmetric system whose architecture consists of a single cycle of size 4. Here, there is no *introduction of asymmetry* as in Table 9.1. The table also has some additional rows for rep_S values of the input and output. These measure the amount of symmetry that is still present in the input or output after the respective number of computation cycles. For the remainder of this subsection, we will formalise this notion.

Round		1	2	3	4	5	6	7	...
Output of the processes	P_0	0	0	1	0	0	1	0	...
	P_1	0	0	1	0	0	0	0	...
	P_2	0	0	1	0	0	1	0	...
	P_3	0	0	1	0	0	0	1	...
Input to P_0		0	1	0	1	0	1	0	...
		0	1	0	0	0	1	0	...
		0	1	0	1	0	0	0	...
		0	1	0	0	0	1	0	...
Input to P_1		0	1	0	0	0	1	0	...
		0	1	0	1	0	0	0	...
		0	1	0	0	0	1	0	...
		0	1	0	1	0	1	0	...
Input to P_2		0	1	0	1	0	0	0	...
		0	1	0	0	0	1	0	...
		0	1	0	1	0	1	0	...
		0	1	0	0	0	1	0	...
Input to P_3		0	1	0	0	0	1	0	...
		0	1	0	1	0	1	0	...
		0	1	0	0	0	1	0	...
		0	1	0	1	0	0	0	...
$\mathrm{rep}_S(\mathit{Output})$		4	4	4	4	4	2	1	...
$\mathrm{rep}_S(\mathit{Input})$		4	4	4	2	2	1	1	...

Table 9.2: Example for correct input/output symmetries for a simple rotation-symmetric architecture with four processes. The rep_S values for the input/output up to the respective rounds are also given.

Definition 37. *Let* AP *be some set, and* $P = \{0, \ldots, n-1\}$ *be a list of process identifiers. For every* $x \subseteq (\mathsf{AP} \times P)$ *and* $w = w_0 w_1 w_2 \ldots w_n \in (2^{\mathsf{AP} \times P})^+$*, we define*

$$\begin{aligned} \mathrm{rep}(x) &= |\{j \in \{0, \ldots, n-1\} \mid \mathrm{rot}(x, j) = x\}| \\ \mathrm{rep}_S(\epsilon) &= n \\ \mathrm{rep}_S(w) &= \gcd(\mathrm{rep}_S(w_0 \ldots w_{n-1}), \mathrm{rep}(w_n)) \end{aligned}$$

In this definition, the operator *gcd* refers to taking the greatest common divisor of two numbers. The definition of the rep_S function above describes how to compute the symmetry degree of a word, i.e., the number of processes getting the same rotations of an input or the number of rotations of the output of the processes that lead to the same element of O. We prove this in two steps and start with the following sub-lemma:

Lemma 5. *If there are precisely* m *values* $j \in \{0, \ldots, n-1\}$ *for an* $m \in \mathbb{N}$ *such that* $\mathrm{rot}(t, j) = t$ *for some* $t \in \mathcal{I}^*$*, then the list of indices* $L = \{\frac{0 \cdot n}{m}, \frac{1 \cdot n}{m} \ldots \frac{(m-1) \cdot n}{m}\}$ *is precisely the list of indices* $\geqslant 0$ *but* $< n$ *such that for all* $l \in L$*:* $\mathrm{rot}(t, l) = t$ *but for all* $l' \notin L$*:* $\mathrm{rot}(t, l') \neq t$ *or either* $l' < 0$ *or* $l' \geqslant n$*.*

Proof. For all $j, j' \in L$, we know that $j + j' \in L$ as well since for all $t \in \mathcal{I}^*$, $\mathrm{rot}(t, j + j') = \mathrm{rot}(\mathrm{rot}(t, j'), j) = \mathrm{rot}(t, j') = t$. Furthermore, $\mathrm{rot}(t, n) = t$.

To show that all elements in $L \cup \{n\}$ are equally spaced (modulo n), consider the converse. So we have $0 \leqslant l < l' < l'' < n$ with $l' - l \neq l'' - l'$ and there are no indices in L in between l' and l or l'' and l', respectively. By the argument above if $l' - l < l'' - l'$ we also have $l' + (l' - l) \in L$ or if $l' - l > l'' - l'$ we also have $l + (l'' - l') \in L$, which is a contradiction. The case that involves wrapping around in the modulo space can be proven similarly.

So we know that there are m equally spaced elements in L and since $\mathrm{rot}(t, 0) = t$ and $\mathrm{rot}(t, n) = t$ for all $t \in \mathcal{I}^*$, the claim follows. □

Lemma 5 can alternatively be shown by applying a theorem by Fine and Wilf (1965) on the combinatorics on words. To use it, we however would have to rearrange the letters in a word, and describing

that construction would be more complicated than giving a direct proof, which is the latter has been done here.

Lemma 6. *For every $t_0 \dots t_{k-1} \in \mathcal{I}^k, k \in \mathbb{N}$, we have*

$$\mathrm{rep}_S(t_0 \dots t_{k-1}) = |\{j \in \{0, \dots, n-1\} : \mathrm{rot}(t_0 \dots t_{k-1}, j) = t_0 \dots t_{k-1}\}|$$

Proof. The proof is done by induction on the length of t.

Basis: Trivial, since $\mathrm{rot}(\epsilon, j) = \epsilon$ for every $j \in \mathbb{Z}$.

Inductive step: Assume that the number of neutral rotations for $t_0 \dots t_{k-2}$ is m (we denote those rotation values j of $t_0 \dots t_{k-2}$ to be *neutral* for which $\mathrm{rot}(t_0 \dots t_{k-2}, j) = t_0 \dots t_{k-2}$) and the number of neutral rotations for $t_0 \dots t_{k-1}$ is m' . By the inductive hypothesis, $\mathrm{rep}_S(t_0 \dots t_{k-2}) = m$.

Clearly, m' is a divisor of m since otherwise there exists a $y \in \{\frac{0 \cdot n}{m'}, \dots, \frac{(m'-1) \cdot n}{m'}\}$ such that $rot(t_0 \dots t_{k-2}, y) \neq t_0 \dots t_{k-2}$, so:

$$rot(t_0 \dots t_{k-1}, y) = rot(t_0 \dots t_{k-2}, y) rot(t_{k-1}, y) \neq t_0 \dots t_{k-2}\, \mathrm{rot}(t_{k-1}, y) = t_0 \dots t_{k-1}$$

Analogously, m' is a divisor of $\mathrm{rep}(t_k)$. In both cases, we would otherwise get a contradiction with Lemma 5.

On the other hand, for every m' that is a divisor of m and $\mathrm{rep}(t_k)$, we have for all $y \in \{\frac{0 \cdot n}{m'}, \dots, \frac{(m'-1) \cdot n}{m'}\}$:

$$\begin{aligned}
&\mathrm{rot}(t_0 \dots t_{k-1}, y) \\
&= \mathrm{rot}(t_0 \dots t_{k-2}, y)\, \mathrm{rot}(t_{k-1}, y) \\
&= t_0 \dots t_{k-2}\, \mathrm{rot}(t_{k-1}, y) && \text{by Lemma 5} \\
&= t_0 \dots t_{k-2} t_{k-1} && \text{by Lemma 5} \\
&= t_0 \dots t_{k-1}
\end{aligned}$$

Clearly, the greatest common divisor of m and $\mathrm{rep}(t_k)$ is the (unique) greatest such number, therefore $\mathrm{rep}_S(t_0 \dots t_{k-1}) = m'$. □

Note that the rep_S function is invariant under rotating its argument (by definition). Thus, by the lemma above, we can immediately deduce the following fact:

Corollary 2. *Let $t \in \mathcal{I}^*$ be the prefix of some input to a rotation-symmetric system with $k = \mathrm{rep}_S(t)$ and $i \in \mathbb{N}$. After $|t|$ computation cycles, the processes $\{P_{(j \cdot \frac{n}{k} + i) \bmod n} \mid j \in \mathbb{N}\}$ have read the same input. For a rotation-symmetric architecture, it thus follows that the processes in these process sets must produce the same output for the first $|t| + 1$ computation cycles.*

From this corollary, we can in turn deduce the main result of this subsection.

Definition 38. *Let $\langle T, \tau \rangle$ be a computation tree. If for every $t = t_0 t_1 \dots t_k \in T$, we have $\mathrm{rep}_S(t_0 t_1 \dots t_k) \mid \mathrm{rep}(\tau(t))$, then we say that $\langle T, \tau \rangle$ does not introduce asymmetry.*

Corollary 3. *If $\langle T, \tau \rangle$ is a computation tree induced by the aggregated Moore machine of a rotation-symmetric architecture, then $\langle T, \tau \rangle$ does not introduce asymmetry.*

9.1.2 Normalisation

Consider the two example (prefix) input streams of a system given in Table 9.3. The input streams given can be unified by rotation (i.e., for them being denoted as w and w', there exists some $j \in \mathbb{N}$ such that $\mathrm{rot}(w, j) = w'$), however the output streams that the system produces when reading the input streams cannot (despite the absence of introduction of asymmetry). This can be seen from the fact that in the third computation cycle, $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ is put out in the one case, but $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is put out in the other case. Such a behaviour would require different implementations of the first process and the second process, as only in this way the two cases can be separated, contradicting the rotation-symmetry of the system.

We define a normalisation function that maps all input streams that can be unified by rotation onto the same input stream. This function will help us later in the actual synthesis algorithm to rule out cases of input/output inconsistencies like the one described above.

Round	1	2	3	4	5	6	7	...
First input	0	1	0	1	0	1	0	...
stream	0	0	0	0	0	1	0	...
First output	0	0	**1**	0	0	0	0	...
stream	0	0	**1**	0	0	0	0	...
Second input	0	0	0	0	0	1	0	...
stream	0	1	0	1	0	1	0	...
Second output	0	0	**0**	0	0	0	0	...
stream	0	0	**0**	0	0	0	0	...

Table 9.3: Two runs of a system that cannot be symmetric

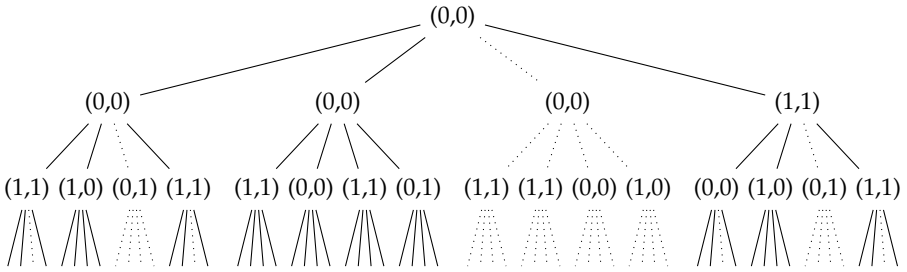

Figure 9.2: A symmetric example execution tree. Tree branches not corresponding to normalised directions are depicted using dotted lines.

For all $t \in \mathcal{I}^*$, we define

$$\eta_S(t) = \min_{i \in \{0,\dots,n-1\}} \mathrm{rot}(t, i)$$

For simplicity, we take the two-dimensional lexicographical minimum of the possible rotations of the input stream as our normalisation function. Two-dimensionality in this context means to choose a rotation such that for the rotated word of the form

$$w = \begin{pmatrix} w_{0,0} \\ w_{0,1} \\ \dots \\ w_{0,n-1} \end{pmatrix} \begin{pmatrix} w_{1,0} \\ w_{1,1} \\ \dots \\ w_{1,n-1} \end{pmatrix} \begin{pmatrix} w_{2,0} \\ w_{2,1} \\ \dots \\ w_{2,n-1} \end{pmatrix} \dots,$$

we have that $w_{0,0}w_{0,1} \dots w_{0,n-1}w_{1,0}w_{1,1} \dots w_{1,n-1}w_{2,0}w_{2,1} \dots$ is lexicographically minimal among all rotations.

This choice of the normalisation function has the advantage to allow normalisation to be done on-the-fly, i. e., we can build a finite-state machine that transforms an input stream to a normalised version of the input stream without delay. Note that we can take a computation tree and strip all branches from it that correspond to non-normalised inputs: as the set of normalised inputs are closed under taking a prefix, the resulting node set is prefix-closed and thus still forms a tree. To illustrate the definition, Figure 9.2 shows an example computation tree for a rotation-symmetric architecture with four processes (and a single cycle) in which all branches that correspond to non-normalised input streams are marked using dotted lines.

Lemma 7. *Let $\langle T, \tau \rangle$ be a computation tree induced by the aggregated Moore machine of a rotation-symmetric architecture. For every $t = t_0 t_1 \dots t_n \in T$ and $i \in \mathbb{N}$ such that $\mathrm{rot}(\eta_S(t), i) = t$, we have $\tau(t) = \mathrm{rot}(\tau(\eta_S(t)), i)$.*

Proof. By the symmetry lemma, its holds that for every computation tree $\langle T, \tau \rangle$ we have $\tau(\mathrm{rot}(t, i)) = \mathrm{rot}(\tau(t), i)$ for all $t \in T$ and $i \in \mathbb{N}$. Since $\eta_S(t)$ is a rotation of t for all $t \in T$, the claim follows. □

The lemma essentially states that the behaviour of a rotation-symmetric system only depends on its behaviour along normalised input streams.

9.1.3 Putting it all together

In the previous two subsections, we have defined necessary conditions for a computation tree to represent the behaviour of a symmetric system, and introduced input normalisation as a tool to filter out computation trees that are inconsistent with respect to rotation. We will now show that these concepts enable us to incorporate the symmetry requirement into a synthesis algorithm.

Definition 39 (Symmetric completion)**.** *Let $\langle T, \tau\rangle$ be a computation tree with $T = \mathcal{I}^*$ and $\tau : T \to \mathcal{O}$. We call a tree $\langle T, \tau'\rangle$ with $\tau' : T \to \mathcal{O}$ a symmetric completion of $\langle T, \tau\rangle$ if for all $t \in T$, we have $\tau'(t) = \mathrm{rot}(\tau(\eta_S(t)), i)$ for some $i \in \mathbb{N}$ with $\mathrm{rot}(\eta_S(t), i) = t$.*

Lemma 8. *Every tree that does not introduce asymmetry has a unique symmetric completion. Trees with the symmetry property are their own symmetric completion.*

Proof. The fact that there exists a symmetric completion for every computation tree is trivial.

Let $\langle T, \tau\rangle$ now be a computation tree that does not introduce asymmetry, and let $\langle T, \tau'\rangle$ be its symmetric completion. For every $t \in T$, there are $\mathrm{rep}_S(t)$ many indices i such that $\mathrm{rot}(\eta_S(t), i) = t$ (Lemma 6). By assumption, we know that there are $\mathrm{rep}_S(\tau(t))$ many indices $i \in \mathbb{N}$ such that $\mathrm{rot}(\tau(\eta_S(t)), i) = \tau(t)$, and $\mathrm{rep}_S(t)$ divides $\mathrm{rep}_S(\tau(t))$. As these index sets share at least one common element and the indices are equally spaced by Lemma 5, the set of these indices for the rotation of t are a subset of the indices for $\tau(t)$. Thus, when choosing an index i for defining $\tau'(t) = \mathrm{rot}(\tau(\eta_S(t)), i)$, it does not matter which one we choose, as we always get the same result. Thus, $\langle T, \tau'\rangle$ is uniquely defined.

For trees with the symmetry property, the fact that they are not altered by symmetric completion follows from the definition of the symmetry property. □

Lemma 9 (Second symmetry lemma)**.** *Let $\langle T, \tau\rangle$ be a computation tree with $T = \mathcal{I}^*$ and $\tau : T \to \mathcal{O}$ for which for every $t \in T$, we have $\mathrm{rep}_S(t) \mid \mathrm{rep}_S(\tau(t))$. The unique symmetric completion of $\langle T, \tau\rangle$ has the symmetry property. Furthermore, if $\langle T, \tau\rangle$ is regular, then so is $\langle T, \tau'\rangle$.*

Proof. The first part of the claim follows directly from the definition of the symmetric completion.

For the second part, we assume that we are given some finite-state machine $\mathcal{M} = (S, \mathcal{I}, \mathcal{O}, s_{init}, \delta, L)$ that induces the computation tree $\langle T, \tau\rangle$. Since this tree is regular, this assumption is valid without loss of generality.

We prove the regularity of $\langle T, \tau'\rangle$ by building a finite-state machine $\mathcal{M}' = (S', \mathcal{I}, \mathcal{O}, s'_{init}, \delta', L')$ that in turn corresponds to $\langle T, \tau'\rangle$.

Recall that the states in $\mathcal{M}$ are the equivalence classes of nodes in $\langle T, \tau\rangle$. Without loss of generality, let us assume that elements $t, t' \in T$ such that $\mathrm{rep}_S(t) \neq \mathrm{rep}_S(t')$ are never put into the same equivalence class. Note that this can only blow up the number of equivalence classes by a factor of at most n.[1] Furthermore assume that the representative element chosen for every equivalence class is a normalised prefix input whenever possible (again, without loss of generality). Thus, for every $s \in S$ representing a normalised input and $i \in \mathcal{I}$, $\min_{u\in\mathbb{N}} \mathrm{rot}(si, u)$ is normalised as well, and so is then $[\min_{u\in\mathbb{N}} \mathrm{rot}(si, u)]$. We construct $\mathcal{M}'$ as follows:

$$
\begin{aligned}
S' &= S \times \{0, \ldots, n-1\} \\
s'_{init} &= (s_{init}, 0) \\
\delta'((s,k), i) &= ([\min_{j\in\mathbb{N}} \mathrm{rot}(\mathrm{rot}(s,k)i, j)], -\operatorname*{argmin}_{j\in\mathbb{N}} \mathrm{rot}(\mathrm{rot}(s,k)i, j)) \\
L'(s,k) &= \mathrm{rot}(L(s), k)
\end{aligned}
$$

Let us now prove that $\mathcal{M}'$ represents the symmetric completion of $\langle T, \tau\rangle$. For this, it suffices to show that for every $t \in T$, we have $\delta'(s_0, t) = ([\eta_S(t)], -\operatorname{argmin}_{j\in\mathbb{N}} \mathrm{rot}(t, j))$. By the definition of L', we then have that $L'(\delta'(s_{init}, t)) = \mathrm{rot}(L(\eta_S(t)), -\operatorname{argmin}_{j\in\mathbb{N}} \mathrm{rot}(t, j))$ (as $L([\eta_S(t)]) = L(\eta_S(t))$), which corresponds to the symmetric completion of $\langle T, \tau\rangle$.

[1] In principle, forcing two tree nodes to not be in the same equivalence class if they do not have some common property can make the number of equivalence classes infinite. Here, this is however not the case as for every $t \in T$ and $x \in \mathcal{I}$, $\mathrm{rep}_S(tx)$ can be computed from $\mathrm{rep}_S(t)$ and x. This way, computing the rep_S values of the prefix input stream can be done on-the-fly by a finite-state machine while reading the input, and when we compute its product with $\mathcal{M}$, the set of states of the resulting finite-state machine is then finite and serves as set of equivalence classes for the tree $\langle T, \tau\rangle$ such that no tree nodes with different rep_S values are put into the same equivalence class.

Let us show that we have $\delta'(s_0, t) = ([\eta_S(t)], -\operatorname{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))$ for every $t \in T$ by induction. The induction basis for $t = \epsilon$ is trivial. For the inductive step, we have (for $t \in T$ and $x \in \mathcal{I}$):

$$\begin{aligned}
\delta'(s_0, tx) &= ([\min_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}([\eta_S(t)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)], \\
&\quad -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}([\eta_S(t)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)) && (9.13)\\
&= ([\min_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}([\min_{l\in\mathbb{N}} \operatorname{rot}(t, l)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)], \\
&\quad -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}([\min_{l\in\mathbb{N}} \operatorname{rot}(t, l)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)) && (9.14)\\
&= ([\min_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}(\min_{l\in\mathbb{N}} \operatorname{rot}(t, l), -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)], \\
&\quad -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(\operatorname{rot}(\min_{l\in\mathbb{N}} \operatorname{rot}(t, l), -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))x, j)) && (9.15)\\
&= ([\min_{j\in\mathbb{N}} \operatorname{rot}(tx, j)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(tx, j)) && (9.16)\\
&= ([\eta_S(tx)], -\operatorname*{argmin}_{j\in\mathbb{N}} \operatorname{rot}(tx, j)) && (9.17)
\end{aligned}$$

The first line in this deduction is obtained by applying the induction hypothesis to the definition of δ. From line (9.13) to line (9.14), we applied the definition of η_S. From line (9.14) to line (9.15), we used the property that by the fact that we are concerned with an equivalence relation over subtrees, we know that for all $t \in T$ and $i \in \mathcal{I}$, we have $[[t]x] = [tx]$. This fact also holds for all rotations of $[t]x$ and tx. From line (9.15) to line (9.16), we get by simplifying $\operatorname{rot}(\min_{l\in\mathbb{N}} \operatorname{rot}(t, l), -\operatorname{argmin}_{j\in\mathbb{N}} \operatorname{rot}(t, j))$ to t, as in this equation, the two rotations even up. The step to the last line just represents using the definition of the η_S function. □

The second symmetry lemma states that whenever we want to synthesise a symmetric system, it suffices to search for a computation tree $\langle T, \tau\rangle$ such that for every $t \in T$, we have $\operatorname{rep}_S(t) \mid \operatorname{rep}_S(\tau(t))$ (so no asymmetry is introduced), and such that its *symmetric completion* to a tree $\langle T, \tau'\rangle$ as defined above satisfies the specification. Note that this approach is complete, as by the fact that every symmetric computation tree is the symmetric completion of itself, it is a possible outcome. Requiring the computation tree $\langle T, \tau\rangle$ to not introduce asymmetry ensures that the symmetric completion is unique and that that the symmetric completion has the symmetry property.

9.1.4 Synthesis of rotation-symmetric systems

We now describe how to use the fact that we can decompose the symmetry property for computation trees into the two sub-properties described earlier in this chapter in an actual synthesis process.

Incorporating the absence of introduction of asymmetry is easy, as this sub-property is regular. Making all branches in the synthesised computation tree depend only on the branches corresponding to normalised inputs is more difficult, and we circumvent this problem by modifying the specification automaton such that it accepts precisely the computation trees whose symmetric completions satisfy the specification. This way, we can apply an ordinary synthesis procedure to obtain some computation tree, and then fix the computation tree to be symmetric. In order to make sure that the original specification is satisfied on the parts of the computation tree that are replaced by the symmetric completion operation, this approach requires instantiating the specification to all possible rotations.

In a nutshell, symmetric synthesis for rotation-symmetric architectures with n processes with some LTL specification ψ over $\mathcal{I}/\mathcal{O}$ can be performed as follows:

1. Using the specification ψ, take $\psi' = \psi \wedge \operatorname{rot}(\psi, 1) \wedge \ldots \wedge \operatorname{rot}(\psi, n-1)$.
2. Translate ψ' to an equivalent universal co-Büchi word automaton $\mathcal{A}$.
3. Compute a deterministic safety word automaton $\mathcal{A}_1$ over $\mathcal{I} \times \mathcal{O}$ that accepts all words for which symmetry is not broken by the output. Take the conjunction of $\mathcal{A}_1$ and $\mathcal{A}$ to obtain an automaton $\mathcal{A}'$.

4. Compute the $\mathcal{I}/\mathcal{O}$-spread of $\mathcal{A}'$. Let it be called $\mathcal{A}''$. Perform a (constructive) emptiness test on $\mathcal{A}''$.

5. If $\mathcal{A}''$ is non-empty, take the symmetric completion of the computation tree and chop off the outputs for all processes except for the first one. The resulting (regular) tree is a valid implementation for one process such that the symmetric product of the tree satisfies the original specification.

The first step of this sequence is trivial. The second step has been described extensively in the literature (Kupferman and Vardi, 2005), and is described in Section 3. In general, when translating an LTL formula to a universal co-Büchi automaton, we get an exponential blowup. Since ψ' is of size $|\psi| \cdot O(n)$, $\mathcal{A}_W$ would normally be of size exponential in n and $|\psi|$. However, as ψ' is basically a big conjunction, and universal co-Büchi automaton blow up only linearly under taking a conjunction, we can translate all conjuncts $\{\mathrm{rot}(\psi, i)\}_{i\in\{0,\dots,n-1\}}$ of ψ to universal co-Büchi automata individually and then take their conjunction on the automaton level to obtain an automaton $\mathcal{A}_W$ of size exponential in $|\psi|$, but only linear in n.

For step 3, we construct $\mathcal{A}_1$ as follows:

Lemma 10. *There exists a deterministic safety word automaton $\mathcal{A}_1 = (Q, \mathcal{I} \times \mathcal{O}, \delta, q_0)$ that accepts precisely the words that do not witness the introduction of asymmetry over $\mathcal{O}$ (with respect to $\mathcal{I}$).*

Proof. We build $\mathcal{A}_1$ as follows.

$$
\begin{aligned}
Q &= \{j \in \mathbb{N} \mid \exists k \in \mathbb{N} : k \cdot j = n\} \\
\delta(q, (i, o)) &= \begin{cases} \{\gcd(q, \mathrm{rep}(i))\} & \text{if } q \mid \mathrm{rep}(o) \\ \varnothing & \text{else} \end{cases} \\
&\qquad \text{for all } q \in Q, (i, o) \in \mathcal{I} \times \mathcal{O} \\
q_0 &= n
\end{aligned}
$$

The construction ensures that the states in an infinite run of $\mathcal{A}_1$ for some (accepted) word represent the rep_S value of the $\mathcal{I}$-part of the input word (by closely following Definition 37).

However, not all runs are accepted. For a deterministic safety automaton, this means that for some input word, the (unique) run of the automaton is finite. Assume that some run is not accepted. Then, at some point, we have that $q \nmid \mathrm{rep}(o)$. In this case, as q describes the rep_S value of the input so far, this corresponds to an introduction of asymmetry.

As all other words are accepted, the automaton performs the task it is declared to perform. □

To take the conjunction between $\mathcal{A}_1 = (Q, \mathcal{I} \times \mathcal{O}, \delta, q_0)$ and $\mathcal{A}$, we translate $\mathcal{A}_1$ to a universal co-Büchi word automaton. This can easily be done by introducing a new state q_{fail} to $\mathcal{A}_1$, adding transitions to q_{fail} for predecessor state/input combinations for which δ has no successor, adding self-loops to q_{fail} for every input symbol, and taking $\{q_{fail}\}$ as the set of rejecting states. After both automata are in form of UCWs, we can simply merge their state sets, transitions, initial state sets, and rejecting state sets. To obtain a UCW that has again only one initial state, we can apply the idea for adapting the automaton from page 34.

For the fourth step, spreading the universal co-Büchi word automaton to a co-Büchi tree automaton is described in Chapter 5.2. Then, the emptiness test described by Kupferman and Vardi (2005) can be used, whose running time is exponential in the number of states of $\mathcal{A}''$. Alternatively, we can apply the *bounded synthesis* approach (Schewe and Finkbeiner, 2007), which is of the same complexity, but lends itself to efficiently implementing it (Ehlers, 2010a, 2011a; Filiot et al., 2009, 2010; Bohy et al., 2012).

Finally, if the emptiness test yields a (regular) tree, it has to be modified in order to obtain a (regular) tree with the symmetry property that satisfies the original specification. This process is described in in the Proof of Lemma 9.

We have now set the scene for the main theorem:

Theorem 10. *Symmetric synthesis for rotation-symmetric architectures can be performed by using the six steps described above in time exponential in the number of processes and in time doubly-exponential in the length of the LTL specification.*

Proof. For the proof, we show three individual claims:

1. Every regular symmetric tree that satisfies the specification is accepted by $\mathcal{A}''$ and is identical to its symmetric completion.
2. The symmetric completion of every accepted tree satisfies the specification.
3. The five steps do not require more time than claimed.

Claim 1: For the first part of the proof, assume that for an execution tree $\langle T, \tau \rangle$ with the symmetry property, every branch satisfies the specification ψ. By Lemma 7, symmetric completion does not alter trees that already have the symmetry property. Additionally, $\langle T, \tau \rangle$ does not introduce asymmetry into its behaviour (since it is symmetric), so every branch is accepted by $\mathcal{A}'$ if and only if every branch is accepted by $\mathcal{A}$.

Note that if a tree with the symmetry property satisfies ψ, then it also satisfies all of ψ's rotations as for each path through $\langle T, \tau \rangle$, we can rotate the input and output by j for some $j \in \mathbb{N}$ and obtain a path that is also in the tree (due to its symmetry property). Since this path satisfies ψ and we can repeat this line of reasoning for all paths, $\mathrm{rot}(\psi, j)$ is satisfied as well. Thus, if every branch of the tree satisfies ψ, they satisfy ψ' as well, and are thus accepted by $\mathcal{A}$ and by $\mathcal{A}'$ as well, and thus $\mathcal{A}''$ accepts the overall tree.

Claim 2: For the other direction, consider a tree $\langle T, \tau \rangle$ that is accepted by $\mathcal{A}''$. By the construction of $\mathcal{A}''$, it satisfies every rotation of ψ on branches corresponding to normalised directions. Using the fact that $\mathcal{A}''$ checks for the introduction of asymmetry (by being the spread of an automaton resulting from a conjunction with $\mathcal{A}_1$, which checks precisely this), by Lemma 9, the symmetric completion of $\langle T, \tau \rangle$ has the symmetry property. Likewise, as symmetrising does not change the branches corresponding to normalised directions and as in a symmetric completion, all other branches are just rotations of normalised ones, they satisfy all rotations of ψ as well, so in particular ψ itself.

Claim 3: After the first three steps of the construction, $\mathcal{A}'$ has a number of states that is exponential in $|\psi|$ but polynomial in n. As $\mathcal{A}''$ is the spread of $\mathcal{A}'$, and spreading the automaton does not lead to a superlinear blow-up (in the number of states, the only blowup occurs in the size of $\mathcal{I}$), the same applies to $\mathcal{A}''$. Since emptiness checking of $\mathcal{A}''$ can be performed in time exponential in the number of states in $\mathcal{A}''$ and linearly in the input/output width (which is exponential in n), we obtain a complexity that is doubly exponential in $|\psi|$ and exponential in n. A finite-state machine that describes the symmetric completion of the tree obtained (if any) can be done in time polynomial in the number of equivalence classes in the tree and n, as described in the proof of Lemma 9. □

9.1.5 Complexity analysis

Let us now prove that the symmetric synthesis algorithm for rotation-symmetric architectures described previously in this chapter is optimal in the complexity-theoretic sense.

Due to the fact that standard (open) LTL synthesis is embedded in the symmetric synthesis problem (by setting $n = 1$), the 2EXPTIME-hardness of standard LTL synthesis (Pnueli and Rosner, 1989a) is inherited in the symmetric setting. Since the construction given in the preceding chapter is also doubly exponential in the length of the specification, it remains to be answered how hard the problem is with respect to the number of components. It turns out that the problem has a complexity that is exponential in n.

We base our hardness proof on general ideas established for synthesis hardness proofs for temporal logics established by Fischer and Ladner (1979) as well as Vardi and Stockmeyer (1985). We reduce the acceptance problem of an alternating space-bounded Turing machine (Chandra et al., 1981) for a given word to the synthesis problem. To ensure that computations that exceed the space available on the tape are taken into account correctly, we build a specification that is realisable if some word is **not** accepted, which differentiates our proof from, e.g., the one for one general CTL* synthesis hardness by Vardi and Stockmeyer (1985).

Definition 40 (Chandra et al., 1981)**.** *An alternating Turing machine is defined by a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, g)$ with the set of states Q, the input alphabet Σ, the tape alphabet Γ, the transition function δ, the initial state q_0, and $g : Q \to \{accept, reject, \vee, \wedge\}$ marking the types of the states. We say that the Turing machine is $f(k)$-SPACE*

bounded for some function $f : \mathbb{N} \to \mathbb{N}$ if for every accepted word of length k, M also accepts the word when considering all runs whose space usage exceeds $f(k)$ to be rejecting.

For the scope of this paper, when discussing $f(k)$-space bounded Turing machines, we only consider functions $f(k)$ that are easy to compute (in time polynomial in $f(k)$). For every accepted word, we can arrange a set of runs of an alternating Turing machine for that word in a run tree that splits whenever a state with universal branching is visited. All runs in this tree also do not exceed the space bound. By definition, for every accepted word, there exists such a tree, and all of its leafs are accepting configurations.

Lemma 11. *Given an $f(k)$-space bounded alternating Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, g)$, we can reduce the acceptance of a word $w \in \Sigma^k$ by M to the symmetric realisability problem of $n = f(k)$ processes with a specification in LTL of size polynomial in $|Q| \cdot |\Gamma| \cdot |w|$.*

Proof. The proof is done by constructing a specification ψ that is realisable in the symmetric setting if and only if $w = w_0 \dots w_{k-1}$ is not accepted. We define $\psi = \psi_1 \wedge \psi_2$ for ψ_1 and ψ_2 to be given below. We have $\mathsf{AP}_I^L = \{C\}$ and $\mathsf{AP}_O^L = \{S^0, \dots, S^{|\mathsf{AP}_O^L|}\}$ of sufficient cardinality to encode every element from the set $\Gamma' = (\Gamma \cup \{\epsilon\} \cup (Q \times \Gamma)) \times 2^{\{\lfloor,\rfloor\}}$. This way, the outputs of the individual processes represent locations on the Turing machine tape. For simplicity, in the following, for all $0 \leqslant j < n$, we denote the set of output atomic propositions for process j by S_j. We also encode the current state of the machine and tape end markers onto the tape. Note that in the symmetric setting we do not have a designated process for the initial state. The formula ψ_1 makes sure that precisely the processes retrieving a C in the first round start a Turing computation (provided that the initial tapes would not collide), so $\psi_1 = (\neg C_{-k+1} \wedge \dots \wedge \neg C_{-1} \wedge C_0 \wedge \neg C_1 \wedge \dots \wedge \neg C_{k-1}) \Rightarrow (S_0 = (q_0, w_0, \{\lfloor\}) \wedge S_1 = (w_1, \varnothing) \wedge \dots \wedge S_{k-2} = (w_{k-2}, \varnothing) \wedge S_{k-1} = (w_{k-1}, \{\rfloor\}))$.

In order to deal with multiple computations starting in the first round, the delimiters $\lfloor$ and $\rfloor$ mark the ends of the tape of a machine. The specification part ψ_2 makes sure that the simulated Turing machine(s) do(es) not accept the input word, i. e. the processes have to simulate the computation on the Turing tape(s) but never reach an accepting state. For this, we let the choice of the next state for existential branching be made by the external input whereas for universal branching, the choice is made by the system.

We syntactically extend LTL slightly for improved readability by allowing Boolean operations over sets of symbols. For example, $S = X(S)$ can be unfolded to $\bigwedge_{i=1}^{|\mathsf{AP}_O^L|}(S_i \Leftrightarrow XS_i)$. Furthermore, $S_{-1}, S, S_{+1} \xrightarrow[a]{\delta} X(S_{-1}), X(S), X(S_{+1})$ for $a \subseteq \{1,2\}$ is true if for the part of the tape represented by $S_{-1}|_\Gamma, S|_\Gamma, S_{+1}|_\Gamma$, the transition from $S|_Q$ can be made such that afterwards S_{-1}, S, S_{+1} is a valid part of the configuration (at the same place on the tape), the next state is included at the respective position on the tape, and we have taken the bth of the two possible transitions for $b \in a$. Furthermore the tape is extended correctly such that the computation is never left of the $\lfloor$ marker or right of the $\rfloor$ marker (as seen from the initial configuration). If two markers collide, the next configuration is obtained by simply removing the state information from the part of the tape. If a rejecting state is reached, this rule is also applied. Note that without loss of generality, we can assume that precisely two transitions are possible in each state. For $\tilde{\Gamma} = (\epsilon \cup \Gamma) \times 2^{\{\lfloor,\rfloor\}}$, we set:

$$
\begin{aligned}
\psi_2 = \mathsf{G}(&S \text{ is in } Q \times \Gamma \times 2^{\{\lfloor,\rfloor\}} \wedge g(S|_Q) = \text{“}\wedge\text{”} \rightarrow \\
&(S_{-1}, S, S_{+1} \xrightarrow[\{1,2\}]{\delta} \mathsf{X}(S_{-1}), \mathsf{X}(S), \mathsf{X}(S_{+1}))) \\
\wedge\, \mathsf{G}(&S \text{ is in } Q \times \Gamma \times 2^{\{\lfloor,\rfloor\}} \wedge g(S|_Q) = \text{“}\vee\text{”} \wedge \neg C \rightarrow \\
&(S_{-1}, S, S_{+1} \xrightarrow[\{1\}]{\delta} \mathsf{X}(S_{-1}), \mathsf{X}(S), \mathsf{X}(S_{+1}))) \\
\wedge\, \mathsf{G}(&S \text{ is in } Q \times \Gamma \times 2^{\{\lfloor,\rfloor\}} \wedge g(S|_Q) = \text{“}\vee\text{”} \wedge C \rightarrow \\
&(S_{-1}, S, S_{+1} \xrightarrow[\{2\}]{\delta} \mathsf{X}(S_{-1}), \mathsf{X}(S), \mathsf{X}(S_{+1}))) \\
\wedge\, \mathsf{G}(&(S \text{ is in } \tilde{\Gamma}) \wedge (S_{-1} \text{ is in } \tilde{\Gamma}) \wedge (S_{+1} \text{ is in } \tilde{\Gamma}) \rightarrow \\
&S = \mathsf{X}(S)) \\
\wedge\, \mathsf{G}(&\neg g(S|_Q) \text{ is accepting})
\end{aligned}
$$

Assume that there exists an accepting tree of M's computations for w. In this case, the environment could set the input to the first process to **true** in the first round and to **false** for the other processes. Hence, there is only one computation of the Turing machine being simulated. By the environment choosing the existential branching according to the acceptance tree it can be assured that eventually an accepting state is reached, which is not allowed by ψ. Therefore ψ is unrealisable in this setting.

On the other hand, if w is not accepted by M, then there exists no tree of accepting runs of M (that do not use more than $f(k)$ space) for w. In this case since the implementation can decide which transition to take in case of universal branching, it can assure that the run simulated by an implementation of ψ either ends in a rejecting state or exceeds the space limit. Since in case of collisions of end markers on the tape, the state information can be removed from the output and the processes can then output their tape contents forever, the specification is trivially fulfilled in this case. The same applies for rejecting states, so ψ is realisable with a symmetric system. □

This reduction can now be used to state the complexity of symmetric synthesis:

Corollary 4. *The symmetric realisability problem (for LTL) has a time complexity that is exponential in n.*

Proof. Given the question whether a word $w = w_0 \dots w_{k-1}$ is in the language defined by some $(c+1)$-EXPTIME $= (c)$-AEXPSPACE problem for some $c \in \mathbb{N}_0$, we can reduce it to the symmetric (co-)realisability problem for an LTL specification of length polynomial in k and with a number of processes (c)-exponential in k. Since by the space hierarchy theorem (Ranjan et al., 1991), the (c)-EXPTIME hierarchy is strict for increasing c, we can conclude that in general, we cannot solve the symmetric realisability problem better than in time exponential in the number of components. The corresponding upper bound on the complexity is induced by the symmetric synthesis algorithm presented previously. □

Note that the form of the LTL specification used in the proof of Lemma 11 only makes little use of the rich expressivity of LTL, and the LTL formula is a simple safety constraint. As a consequence, the hardness proof can also be used for other specification logics, such as computation tree logic (CTL, Emerson and Clarke, 1982) and we thus obtain a synthesis complexity that is exponential in the number of processes for this logic as well.

9.1.6 Symmetric synthesis from branching-time specifications

We have only been concerned with the synthesis of rotation-symmetric systems from *linear-time specifications* so far. With this specification class, we cannot express that "something good shall be possible to happen", e.g., that there should be *some* input sequence that triggers a certain output signal. However, if we can describe under which conditions the said output signal should be triggered, the specification can typically be represented as a linear-time property again. As the designer of a system usually has such triggers for the system behaviour in mind, linear-time specifications are often deemed to be sufficient for synthesis, and thus, modern synthesis tools like Ratsy (Bloem et al., 2010b), Acacia+ (Bohy et al., 2012), or Unbeast (Ehlers, 2011a) are only concerned with linear-time specifications. Nevertheless, for some applications, support for branching-time specifications in (symmetric) synthesis is useful.

As an example, consider the *partial design verification* problem of an unfinished distributed symmetric communication protocol. In this context, we ask if the protocol can be completed in a way such that its specification is met. We encode both the overall specification and the part of the protocol already designed into a new specification, and apply a synthesis procedure to check if the protocol can be completed such that it meets the original specification (which might be incomplete). Branching-time properties naturally occur in this context as not all conditions for certain actions in the protocol might have already been fixed, and thus existential quantification over paths in the synthesized computation tree is necessary. Proving a resulting specification to be unrealisable would then show early in the design process of such a protocol that no choice for the trigger for "something good" can make the symmetric protocol work correctly, and thus, the protocol design is already flawed although it is not even finished at that point.

In the following, we support such applications by giving a modification of the symmetric synthesis construction from the previous subsections that is suitable for branching-time specifications.

Definition 41 (Symmetric synthesis preparation)**.** *Let $\mathcal{A} = (Q, \mathcal{I}, \mathcal{O}, \delta, q_{init}, \mathcal{F})$ be an alternating Rabin tree automaton. We define its* rotation-completed version *to be an alternating Rabin tree automaton such that $\mathcal{A} = (Q', \mathcal{I}, \mathcal{O}, \delta', q'_{init}, \mathcal{F}')$ with:*

$$
\begin{aligned}
Q' &= Q \times \{0, \dots, n-1\} \cup \{start\} \\
q'_{init} &= start \\
\mathcal{F}' &= \{(E \times \{0, \dots, n-1\}, F \times \{0, \dots, n-1\}) \mid (E, F) \in \mathcal{F}\} \\
\delta'(start, y) &= \bigwedge_{i \in \{0,\dots,n-1\}} \psi(i), \text{where } \psi(i) \text{ is obtained from } \delta(q_{init}, \mathrm{rot}(y, i)) \text{ by} \\
&\quad \text{replacing every occurrence of some element } (q, x) \text{ for some } q \in Q \text{ and} \\
&\quad x \in \mathcal{I} \text{ by } ((q, i), \mathrm{rot}(x, i)), \text{ for all } y \in \mathcal{O} \\
\delta'((q, i), y) &= \psi(i), \text{where } \psi(i) \text{ is obtained from } \delta(q, \mathrm{rot}(y, i)) \text{ by replacing} \\
&\quad \text{every occurrence of some element } (q', x) \text{ for some } q' \in Q \text{ and} \\
&\quad x \in \mathcal{I} \text{ by } ((q', i), \mathrm{rot}(x, i)), \text{ for all } y \in \mathcal{O}, i \in \{0, \dots, n-1\}, \text{ and } q \in Q
\end{aligned}
$$

Lemma 12. *Let $\mathcal{I}$ be the global input and $\mathcal{O}$ be the global output of of a single-cycle rotation-symmetric architecture with n processes, and $\mathcal{A} = (Q, \mathcal{I}, \mathcal{O}, \delta, q_{init}, \mathcal{F})$ be an alternating Rabin tree automaton. We have that for every tree $\langle T, \tau \rangle$ with $T = \mathcal{I}^*$ and $\tau : T \to \mathcal{O}$ with the symmetry property, if and only if $\langle T, \tau \rangle$ is accepted by $\mathcal{A}$, the tree is accepted by the rotation-completed version of $\mathcal{A}$.*

Proof. The construction in Definition 41 replicates all states in the alternating automaton n times. Every state (q, i) in $\mathcal{A}'$ is essentially the same as state q in $\mathcal{A}$, only that all input and output is rotated by i before interpreting them. The added initial state simulates q_{init}, but branches conjunctively into all possible rotations.

Assume that $\mathcal{A}'$ accepts a tree $\langle T, \tau \rangle$ with the symmetry property. In this case, the tree is accepted by $\mathcal{A}$ as well, as $\mathcal{A}'$ contains a non-rotated version of $\mathcal{A}$ that needs to accept the tree in order for $\mathcal{A}'$ to accept the tree.

Now assume that $\mathcal{A}$ accepts a tree $\langle T, \tau \rangle$ with the symmetry property. Since $\langle T, \tau \rangle$ is rotation-symmetric, a copy of $\mathcal{A}$ that reads all directions and all outputs rotated by some value $i \in \mathbb{N}$ accepts the tree as well, as the tree is invariant under such rotations. Since this line of reasoning applies for all $i \in \mathbb{N}$, $\langle T, \tau \rangle$ is accepted by $\mathcal{A}'$ as well. □

The construction from Definition 41 serves the same purpose as step 1 of the LTL symmetric synthesis algorithm in Section 9.1.4. Let the resulting automaton be called $\mathcal{A}'$. Unlike for symmetric synthesis from LTL properties, it is not ensured that the symmetric completion of a tree accepted by $\mathcal{A}'$ is accepted by either $\mathcal{A}$ or $\mathcal{A}'$.

To see this, consider the specification "there should be some node $t \in T$ in the tree with $\tau(t) \neq \varnothing$". We can easily write down some automaton $\mathcal{A}$ for this specification and then apply Definition 41, which leaves the language of the automaton unchanged. The resulting automaton accepts a tree that has precisely one node $t \in T$ in the tree with $\tau(t) \neq \varnothing$, but for which t is not normalised. If we now take the symmetric completion of this tree, we get that for all $t \in T'$, we have $\tau'(t) = \varnothing$ for the resulting tree $\langle T', \tau' \rangle$, and thus the tree is not accepted any more.

To solve this problem, we need to "bend" transitions in the automaton $\mathcal{A}'$ to observe into normalised directions whenever appropriate. We say that a node in the tree is normalised if it refers to an input sequence that is normalised. Likewise, we say that a direction from some node is normalised if the node reached under that direction corresponds to a normalised input sequence. Whenever we are at a position $t \in T$ in the tree and at a state q in the automaton, and we are about to branch with some state q' to some node $tx \in T$ in the tree that is not normalised, then we can branch to some tree node $\eta_S(tx)$ instead, and examine some rotated output in the future. As in a tree with the symmetry property, the labels in the sub-tree induced by node tx have to be rotations of the ones for node $\eta_S(tx)$, we are in this way able to read the nodes in the symmetric completion of a tree without taking the symmetric completion first.

Definition 42. *Let $\mathcal{A}' = (Q', \mathcal{I}, \mathcal{O}, \delta', q'_{init}, \mathcal{F}')$ be an alternating Rabin tree automaton. We define its* normalised version *to be $\mathcal{A}'' = (Q'', \mathcal{I}, \mathcal{O}, \delta'', q''_{init}, \mathcal{F}'')$ with:*

$$
Q'' = Q' \times \{0, \dots, n-1\} \times \{0, \dots, n-1\}
$$

$$
\begin{aligned}
q''_{init} &= (q'_{init}, n, 0)\\
\mathcal{F}'' &= \{(E \times \{0,\dots,n-1\} \times \{0,\dots,n-1\},\\
&\qquad F \times \{0,\dots,n-1\} \times \{0,\dots,n-1\}) \mid (E,F) \in \mathcal{F}\}\\
\delta''((q,k,j),y) &= \delta'(q, \mathrm{rot}(y,-j)), \textit{where every occurrence of a tuple}\\
&\quad (q',x) \textit{ for some } q' \in Q', x \in \mathcal{I} \textit{ is replaced by}\\
&\quad ((q', gcd(k, \mathrm{rep}(x)), \operatorname*{argmin}_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(x,-i)),\\
&\quad \min_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(x,-i))\\
&\quad \textit{for all } (q,n,k) \in Q'', x \in \mathcal{I}, \textit{ where we assume that}\\
&\quad \operatorname*{argmin}_{i \in K} \psi(i) \textit{ for some } K \textit{ and } \psi \textit{ always returns the minimal element}\\
&\quad i \in K \textit{ that leads to a minimal } \psi(i).
\end{aligned}
$$

Lemma 13. *Let $\mathcal{A}$ be an alternating Rabin tree automaton, $\mathcal{A}'$ be the corresponding rotation-completed version, and $\mathcal{A}''$ be the normalised version of $\mathcal{A}'$. We have that $\mathcal{A}''$ accepts a computation tree if and only if $\mathcal{A}$ accepts its symmetric completion.*

Proof. First of all, note that along a branch in the run tree for a given input tree $\langle T, \tau \rangle$, the second component of the state tuple always tracks the rep_S value of the current tree node (by following the recursive definition of rep_S in the definition of δ''). Furthermore, the third component always represents $\operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i)$ for every tree node t. This can be proven by induction. For $t = \epsilon$, the claim is trivial. For all other $tx \in T$, we have that:

$$
\begin{aligned}
&\operatorname*{argmin}_{i \in \mathbb{N}} \mathrm{rot}(tx,-i) && (9.18)\\
=\ &\operatorname*{argmin}_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(tx,-i) && (9.19)\\
&\text{for } j = \operatorname*{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i) \text{ and } k = \mathrm{rep}_S(t)\\
=\ &\operatorname*{argmin}_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(x,-i) && (9.20)\\
&\text{for } j = \operatorname*{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i) \text{ and } k = \mathrm{rep}_S(t)\\
& && (9.21)
\end{aligned}
$$

From line (9.18) to line (9.19), we apply the fact that that the minimum we are concerned with is a lexicographical minimum, so to the first t elements of the tree node sequence must remain unchanged. The next transformation uses the fact that only rotations under which t is invariant are contained in the set of rotations $\{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}$. Note that as in a run tree of $\mathcal{A}$, the second component always represents the rep_S value of the input corresponding to the run tree branch so far, and by the inductive hypothesis, the third component represents $\operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i)$ for the input so far t, we know that the expression $\operatorname{argmin}_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(x,-i)$ in the definition of δ computes $\operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(tx,-i)$.

After these preparations, let us now prove that for every rotation-symmetric tree $\langle T, \tau \rangle$ and the corresponding run tree $\langle T_r, \tau_r \rangle$ for $\mathcal{A}'$, there exists a run tree $\langle T'_r, \tau'_r \rangle$ for the symmetric completion $\langle T', \tau' \rangle$ of $\langle T, \tau \rangle$ and $\mathcal{A}''$. Furthermore, $\langle T_r, \tau_r \rangle$ is accepting if and only if $\langle T'_r, \tau'_r \rangle$ is accepting.

We construct $\langle T'_r, \tau'_r \rangle$ inductively from the nodes in $\langle T_r, \tau_r \rangle$, and let $\langle T'_r, \tau'_r \rangle$ have the same structure as $\langle T_r, \tau_r \rangle$ by setting $T'_r = T_r$. For every node $t_r \in T_r$ with $\tau_r(t_r) = (q,t)$, we create a node $t_r \in T'_r$ with $\tau'_r(t_r) = ((q, \mathrm{rep}_S(t), \operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i)), \eta_S(t))$.

We start by setting $\tau'_r(\epsilon) = ((q,n,0),\epsilon)$ for $\tau_r(\epsilon) = (q,\epsilon)$. All other nodes are defined inductively. Let $t_r \in T_r$ with $\tau(t_r) = (q,t)$ be a node for which $\tau'_r(t_r) = ((q,k,j),t') = ((q, \mathrm{rep}_S(t), \operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i)), \eta_S(t))$ is already ensured. Let $\Theta \subseteq Q \times \mathcal{I}$ be such that for all $(q,x) \in Q \times \mathcal{I}$, we have $(q,x) \in \Theta$ if and only if for some $m \in \mathbb{N}$, $\tau(t_r m) = (q, tx)$. We know that $\Theta \models \delta(q, \tau(t))$ since $\langle T_r, \tau_r \rangle$ is a run tree of $\langle T, \tau \rangle$. Furthermore, let $\Theta' = \{((q, \mathrm{rep}_S(tx), \operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(tx,-i)), \min_{i \in \{j, j+k, j+2k, \dots, j+(n-1)k\} \cap \{0,\dots,n-1\}} \mathrm{rot}(x,-i))) \mid (q,x) \in \Theta\}$. By the definition of δ' and the rotation-symmetry of $\langle T, \tau \rangle$, we know that $\Theta' \models \delta''((q,k,j), \mathrm{rot}(\tau(t),-j))$. Then, by setting $\tau'_r(t_r m) = ((q, \mathrm{rep}_S(tx), \operatorname{argmin}_{i \in \mathbb{N}} \mathrm{rot}(t,-i)), \eta_S(tx))$ for every m such that $t_r m \in T_r$, we

build the $\langle T'_r, \tau'_r \rangle$-successors of t_r. The resulting run tree $\langle T'_r, \tau'_r \rangle$ is accepting if and only if $\langle T_r, \tau_r \rangle$ is accepting, as for every node t_r, τ_r maps t_r to an accepting state if and only if τ'_r maps t_r to an accepting state.

Proving the other direction can be done analogously, with the main difference that instead of re-routing the directions for which input is considered, this re-routing is undone, using the fact that the input computation tree has the symmetry property. □

Lemma 13 tells us how we can perform symmetric synthesis for branching-time specifications: we translate our specification to an alternating Rabin tree automaton $\mathcal{A}$, apply the constructions from Definition 41 and then Definition 42, and check the final tree automaton for emptiness of its language. If it is not empty, we can continue as in the linear-time case, i. e., we take its symmetric completion and chop off all output except for the first process.

In terms of complexity, starting from an alternating Rabin tree automaton with m Rabin pairs and k states as automaton $\mathcal{A}$, the automaton $\mathcal{A}''$ that we get after applying the constructions from Definition 41 and Definition 42 is of size $O(k \cdot n^3)$ and still has m Rabin pairs. We can then translate this automaton to a non-deterministic Rabin tree automaton and finally check the resulting automaton for emptiness (Muller and Schupp, 1995; Emerson and Jutla, 1988; Kupferman and Vardi, 1997). The last two steps can be performed in time exponential in the number of states of $\mathcal{A}''$ and exponential in m.

Let us now have a look at the symmetric synthesis complexities for the commonly used branching-time logics CTL and CTL*. We can translate a CTL formula to a one-acceptance-pair alternating Büchi tree automaton of size polynomial in the length of the formula (Vardi, 1997). Overall, we obtain a symmetric synthesis complexity that is exponential in the length of the CTL formula (which is optimal - see, e.g., Kupferman and Vardi, 1997), and exponential in the number of processes, which is again optimal, as shown in Chapter 9.1.5.

For CTL*, the algorithm described above is also complexity-theoretically optimal. We can translate a CTL* formula to an alternating Rabin tree automaton with a number of states that is exponential in the length of the formula, and two Rabin pairs (Bernholtz et al., 1994; Kupferman and Vardi, 1997). We can then perform synthesis in time exponential in the number of states of this automaton, and exponential in n. Again, we get an optimal complexity (in n) with our construction, and the 2EXPTIME CTL* synthesis hardness (Vardi and Stockmeyer, 1985) is also matched by our algorithm.

9.2 Tackling general rotation-symmetric architectures

Previously in this chapter, we have only dealt with rotation-symmetric architectures that have a single cycle so far. The only reason for this choice was to simplify the presentation. Instead of working on group elements of the rotation-symmetric group on which the symmetric architecture is based, we were able to describe the rep_S value of an input or output prefix stream by simple integer values. Let us discuss how to lift the ideas so far to the general rotation-symmetric case by means of an example.

Consider the 2x2 symmetric architecture depicted in Figure 8.7. The architecture is basically the product of two cycles of processes. One of these cycles permutes the first two inputs, whereas the other cycle permutes the other inputs. Table 9.4 describes the local inputs that the processes read for one example input stream to the system. Process $P_{0,0}$ reads the input stream in a non-permuted manner, whereas the other processes obtain a permutation of the input stream. In the first two computation cycles, all input bits have the same values. So up to the third computation cycle, all of the output bit values of the processes have to be the same (recall that we assume a Moore-type computation model in this section).

In the third computation cycle, every process receives only one 0 and three 1s. Since however the first two and the second two local inputs of the processes are cycled individually, processes P_0 and P_1 receive the same input, and processes P_2 and P_3 obtain the same input. Thus, symmetry is only broken *partially* at that point. The fact that in the fourth computation cycle, the outputs of P_0 and P_1 then differ from the outputs of P_2 and P_3 does not witness the introduction of asymmetry in the output. In a single-cycle symmetric architecture, the input would have already broken the symmetry completely at that point.

In the sixth computation cycle, the input to the system is then such that every process gets a different local input. At this point, symmetry is fully broken by the input.

For the general case of having more than one cycle in a rotation-symmetric system, we thus have to adapt the definition of the rep_S function (Definition 37) as follows: Let G be our rotation-symmetric

Round		0	1	2	3	4	5	6	...
Output of the processes	$P_{0,0}$	0	0	1	0	0	0	0	...
	$P_{0,1}$	0	0	1	0	0	0	0	...
	$P_{1,0}$	0	0	1	1	0	0	1	...
	$P_{1,1}$	0	0	1	1	0	0	0	...
Input of process $P_{0,0}$		0	1	1	0	0	1	0	...
		0	1	1	0	0	0	0	...
		0	1	0	0	0	0	0	...
		0	1	1	0	0	1	0	...
Input of process $P_{0,1}$		0	1	1	0	0	0	0	...
		0	1	1	0	0	1	0	...
		0	1	0	0	0	0	0	...
		0	1	1	0	0	1	0	...
Input of process $P_{1,0}$		0	1	1	0	0	1	0	...
		0	1	1	0	0	0	0	...
		0	1	1	0	0	1	0	...
		0	1	0	0	0	0	0	...
Input of process $P_{1,1}$		0	1	1	0	0	0	0	...
		0	1	1	0	0	1	0	...
		0	1	1	0	0	1	0	...
		0	1	0	0	0	0	0	...

Table 9.4: Example input and output for the 2x2 rotation-symmetric architecture in Figure 8.7.

group that is obtained by computing $G^1 \times \ldots \times G^k$, where $\times$ denotes the group direct product, and $G^1, \ldots, G^k$ are cyclic groups. For every group G^i, let $\mathcal{E}_g(G^i)$ be a generating element and $\mathcal{E}_n(G^i)$ be a neutral element of the group. We define:

Definition 43. *Let* AP *be some set, and* $P = \{u \mid u \in G\}$ *be a list of process identifiers. For every* $x \subseteq (\mathsf{AP} \times P)$ *and* $w = w_0 w_1 w_2 \ldots w_n \in (2^{\mathsf{AP} \times P})^+$*, we define*

$$\begin{aligned}
\mathrm{rep}(x) =& \{i \mapsto |\{j \in \{0, \ldots, |G_i| - 1\} \mid (\mathcal{E}_n(G^1), \ldots, \mathcal{E}_n(G^{i-1}), (\mathcal{E}_g(G^i))^j, \\
& \qquad \mathcal{E}_n(G^{i+1}), \ldots, \mathcal{E}_n(G^k))(x) = x\} : i \in \{1, \ldots, k\}\} \\
\mathrm{rep}_S(\epsilon) =& \{i \mapsto |G^i| : i \in \{1, \ldots, k\}\} \\
\mathrm{rep}_S(w) =& \{i \mapsto \gcd(\mathrm{rep}_S(w_0 \ldots w_{n-1})(i), \mathrm{rep}(w_n)(i)) : i \in \{1, \ldots, k\}\}
\end{aligned}$$

With this new definition, the rotation symmetry of a prefix word is measured for every cyclic subgroup of our rotation-symmetric group that represents the architecture separately. The definition of η_S for our symmetric synthesis algorithm is adapted in a similar manner. We only need to fix an order of the group elements for taking the lexicographic minimum of the input stream. Whenever we range over $\{0, \ldots, n-1\}$ in some formula for the symmetric synthesis construction then, we need to replace this set by G, as the set of possible rotations does not form a single cycle then any more. This also simplifies the definition of the function rot, as it is now just the application of a group element in G.

9.3 Bounded delay

So far, we have always assumed that all input arrives at all processes synchronously once it is produced. In applications in which the data is produced in a distributed fashion, it is typically transmitted through some communication network, which may cause delays. For these applications, the assumption of synchronous arrival is not always justified. This is no problem for the techniques in this paper, however, as we can require a process to announce its behaviour a few steps in advance in a way such that the behaviour can only depend on the information that the process has available at a certain point in time. A system is realisable with these announcements in the classical symmetric synthesis setting if and only if it is realisable under delays without the announcements, and implementations that realise a specification can be translated between these two settings.

The technique that we present in the following for achieving this is closely related to the one given by Gastin et al. (2009), where a non-deterministic tree automaton construction is used to check for a system whether its behaviour does not depend on the input not yet available. Whenever information arrives with some delay d, they guess the behaviour of the system in the next d rounds non-deterministically in a way that does not depend on information not yet available, and then verify the guess. However, the construction here is simpler, as we can handle the added requirement on the logical specification level, rather than on the tree automaton level. Furthermore, for Gastin et al.'s technique, the use of non-deterministic tree automata is essential, and thus, cannot easily be combined with the currently most widely used approaches for synthesis from linear-time specifications that circumvent the use of non-deterministic tree automata, such as bounded synthesis (Schewe and Finkbeiner, 2007) and GR(1) synthesis (Piterman et al., 2006).

The technique presented here is not bound to be used in the symmetric synthesis setting, but can be used for bounded-delay synthesis also in the monolithic case. To simplify the presentation, we show how it can be used in the latter case and then discuss its application to the symmetric synthesis domain.

Definition 44. *Let $\mathsf{AP} = \{x_0, \ldots, x_{n-1}\}$ be a set of atomic propositions and $d : \mathsf{AP} \to \mathbb{N}$ be a function associating to every atomic proposition some delay. We call $d_{max} = \max_{x \in \mathsf{AP}} d(x)$ the maximum delay of d. Furthermore, for some constant $c' \leqslant d_{max}$, we call a function $f : ((\{(x,i) \in \mathsf{AP} \times \mathbb{N} \mid d(x) \leqslant i < c'\}) \to \mathbb{B}) \to \mathbb{B}$ a delay-lookup table for AP and d.*

Intuitively, a c'-delay-lookup table represents, for some output atomic proposition of a system, under what next c' input valuations of the input atomic propositions AP, the selected output atomic proposition will be set to **true** after the c' computations cycles. These next inputs are represented as functions from combinations of atomic propositions and computation cycles to a Boolean value representing whether the atomic proposition is present. Some combinations are however not in the domain of the function, namely those for which the delay leads to the fact that they are not present early enough to be taken into account for the respective output (i.e., $d(x)$ computation cycles before a decision is to be made).

To force a system to never make decisions on its output values depend on signal valuations before these have arrived, we can now integrate delay-lookup tables in a synthesis process in a simple manner. For every output atomic proposition and every $c' \leqslant d_{max}$, the system has one delay-lookup table that represents how the system will behave in the next c' steps for this output proposition. The structure of the table ensures that only information that is present early enough can be used by the system. We add enough additional output atomic propositions to the system to allow it to always output the current delay-lookup tables, and take the conjunction of our original specification with the constraint that the system behaves according to the tables. The system can freely choose the initial contents of the tables and whenever a new input atomic proposition valuation is read, the constraints ensure that the system has to behave according to the content of the table for $c' = 1$, and the information in the tables for higher values of c' is shifted to the tables with lower c' values using the current valuations of the input atomic propositions. The shifting process formally works as follows:

Definition 45. *Let $\mathsf{AP} = \{x_0, \ldots, x_{n-1}\}$ be a set of atomic propositions, $d : \mathsf{AP} \to \mathbb{N}$ be a delay function and f be some delay-lookup table for some constant $c > 1$. For some $X \subseteq \mathsf{AP}$, we define the X-successor of f as a delay-lookup table for the constant $c - 1$ with $succ(f, X) = \{g \mapsto f(g \cup \{(x, c) \mapsto (x \in X) \mid x \in \mathsf{AP}, d(x) \leqslant c\}) \mid g \in ((\{(x,i) \in \mathsf{AP} \times \mathbb{N} \mid d(x) \leqslant i < c'\}) \to \mathbb{B})\}$.*

Lemma 14. *Let $\mathsf{AP} = \{x_0, \ldots, x_{n-1}\}$ be a set of atomic propositions, $d : \mathsf{AP} \to \mathbb{N}$ be a delay function, and c be some constant. Encoding a c-delay-table over AP and d requires 2^k variables for $k = \sum_{x \in \mathsf{AP}} \max(c - d(x) + 1, 0)$.*

Encoding the requirement for the succession of two delay tables as in Definition 45 into linear-time temporal logic can be done by writing a formula of the form $\mathsf{G}\phi$, where the only temporal operator in ϕ is X, and no nesting of temporal operators is needed in ϕ. The size of the LTL formula is linear in the number of atomic propositions used for encoding the tables.

Theorem 11. *Let $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ be an interface of a reactive system, ψ be a specification over $\mathsf{AP}^I \uplus \mathsf{AP}^O$, and $d : \mathsf{AP}^I \to \mathbb{N}$ represent a function stating with which delays the process receives the individual input signals.*

There exists an implementation for ψ over the interface $\mathcal{I}$ that works under the delays imposed by d if and only if there exists an implementation satisfying $\psi \wedge \psi'$ over the interface $\mathcal{I}' = (\mathsf{AP}^I, \mathsf{AP}^O \cup \mathsf{AP}^T)$, where AP^T are the atomic propositions needed to store the delay-tables for $c' \in \{0, \ldots, d_{max}\}$, and ψ' is an LTL specification stating that the delay-tables evolve as defined in Lemma 14. Furthermore, an implementation for the setting under delay can be obtained from the one for $\psi \wedge \psi'$.

Proof. $\Rightarrow$: Assume that there exists an implementation for ψ and $\mathcal{I}$ that works correctly under the input signals arriving delayed as specified by d. Then, we can transform the implementation to one that receives the inputs instantaneously but maintains (and outputs) delay-tables by always filling into the delay tables the behaviour of the system in the near future, but taking any input $x \in AP_I$ as arriving $d(x)$ computation cycles earlier. Apart from the delay, the behaviour of the system stays the same.

$\Rightarrow$: Assume that there exists an implementation for $\psi \wedge \psi'$ and $\mathcal{I}'$. Then, we can build an implementation for ψ and $\mathcal{I}$ as follows: our new implementation for ψ stores the state of the implementation for $\psi \wedge \psi'$ that it had d_{max} steps ago, and the last d_{max} inputs to the system. Whenever reading a new input character, it has all the information available to update this state. At the same time, the available information is used to trace which delay-tables the implementation for $\psi \wedge \psi'$ would have produced since then. As all input needed to choose the correct output for the current computation cycle is present in the state of the implementation for $\psi \wedge \psi'$, the implementation for ψ knows which output to produce. As the new implementation behaves like the old one, except for the fact that it works under the delays, correctness is assured. □

While this theorem above is concerned with single-process synthesis, it can equally be applied to the symmetric synthesis case, as the techniques are orthogonal. We would first translate the global specification to the one in which we require every process to output its near-future behaviour, and then apply symmetric synthesis as before. The resulting implementation for one process is then translated back as described in the proof of Theorem 11.

Chapter 10

Conclusion & summary

In this part, we performed the first thorough analysis of the problem of synthesising symmetric systems.

We started by defining symmetric architectures and what it means for the synthesis problem for a symmetric architecture to be decidable or not. Then, we established the driving factors for the decidability or undecidability of symmetric architectures.

We proved that even a very simple architecture that only consists of two processes with one input bit and one output bit each has an undecidable synthesis problem if the output of one process is read by the other process. This so-called *S0 architecture* is the simplest architecture in which inter-process communication is present, and thus shows a large class of architectures to have undecidable synthesis problems. To analyse the S0 architecture, we introduced the concept of *word compression*, which allowed us to transfer the undecidability of synthesis for a larger architecture (the *S2 architecture*) to this simpler case. For showing the latter to be undecidable, the main idea was to construct a scenario that ensures that processes cannot get enough information about their identity to break symmetry and then reduce the (undecidable) distributed synthesis problem for the A0 architecture as defined by Pnueli and Rosner (1990) to this case.

We then identified another class of symmetric architectures with undecidable synthesis problems. It consists of the architectures without communication between the processes, but for which we can find an *information fork* in the processes, i.e., a pair of processes that are incomparably informed.

After these negative results, we then examined the class of *rotation-symmetric architectures*. These comprise many distributed system designs that are well-suited for practical applications (see, e.g., Figure 8.8, Figure 8.9, or Chapter 16.2) and we proved them to have decidable synthesis problems for specifications in the temporal logics LTL and CTL*. Decidability was established by giving suitable algorithms for both the linear-time and branching-time cases, which also have optimal complexities.

The key idea of our synthesis algorithm was to synthesise a system that represents the parallel composition of all processes at the same time, and while doing this, to impose the requirement that the generated computation tree shall have the *symmetry property*. To circumvent the problem that the symmetry property is non-regular and can thus not simply be added as a part of the specification, we decomposed the property into an ω-regular property, and a non-ω-regular one. For linear-time properties, the latter can be incorporated into the synthesis approach by modifying the specification, and repairing the computed implementation in case of realisability. For branching-time specifications, we proposed to re-route directions in the tree automaton for the specification to implement the non-ω-regular requirement to the synthesised system, and we need the same repairing step as for the linear-time case.

Both synthesis approaches have a complexity that is exponential in the number of processes. Our EXPTIME-hardness proof shows that this is optimal.

We finished the discussion of symmetric synthesis by showing how our synthesis algorithm can be applied to cases in which the symmetric architecture under concern is rotation-symmetric, but the processes get their inputs in a delayed fashion.

The beauty of the symmetric synthesis approach presented in this part of the thesis is the fact that we can now automatically test the symmetric distributed realisability of many protocols and applications whose previous analyses were completely manual and cumbersome. A well-known example is ensuring mutual exclusion to a resource in a distributed fashion (Lann, 1977). Classical algorithms for this purpose use process identifiers, random numbers, or non-determinism in the transmission delays of messages between the processes to break symmetry. Using symmetric synthesis, we can automate the process of checking whether we can actually construct a mutual exclusion algorithm for a given setting without the need to have symmetry broken by such means.

Part III

Efficient synthesis

Chapter 11

An analysis of generalised reactivity(1) synthesis

Generalised reactivity(1) synthesis (Piterman et al., 2006) is a synthesis approach that is typically introduced as having a complexity that is polynomial in the state space of the design, while the size of the state space is at most exponential in the number of input and output signals of the system under design. Formally, for a system interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$, the supported specifications are of the form

$$(a_1 \wedge \ldots \wedge a_m) \rightarrow (g_1 \wedge \ldots \wedge g_n),$$

where $a_1, \ldots, a_m$ are the *assumptions* that the environment of the system should satisfy, and $g_1, \ldots, g_n$ are the *guarantees*. Every assumption is of one of the forms

(1) ψ_I,

(2) $\mathsf{G}(\psi \rightarrow \mathsf{X}(\psi_I))$, or

(3) $\mathsf{GF}(\psi)$,

where ψ is an LTL formula over $\mathsf{AP}^I \cup \mathsf{AP}^O$ that is free of temporal operators and ψ_I is an LTL-formula over AP^I that is free of temporal operators. Likewise, all guarantees are of one of the forms

(1) ψ,

(2) $\mathsf{G}(\psi \rightarrow \mathsf{X}(\psi'))$, or

(3) $\mathsf{GF}(\psi)$,

where ψ' is an LTL-formula over $\mathsf{AP}^I \cup \mathsf{AP}^O$ that is free of temporal operators.

Constraints of type (1) are used to describe how the system and the environment behave initially when starting their service (*initialisation properties*), while constraints of type (2) describe the intended evolution of the system and environment states (the *basic safety properties*). Constraints of the third type are called *basic liveness properties*.

Generalised reactivity(1) synthesis is always used with Mealy-type semantics. This allows us to encode a large set of properties that are not directly representable as assumptions or guarantees of the forms (1)-(3) in an easy manner. Let us take a look at such an encoding by means of an example.

Example 18. Consider a two-client arbiter system that implements a mutual-exclusion protocol. We have $\mathsf{AP}^I = \{r_1, r_2\}$ for the requests, and $\{g_1, g_2\} \subseteq \mathsf{AP}^O$ for the grants. As (original) specification, we have $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1) \wedge \mathsf{G}(r_2 \rightarrow \mathsf{F}g_2) \wedge \mathsf{G}(\neg g_1 \vee \neg g_2)$. The specification thus has no assumptions, but none of the guarantees is given in the required form. Transforming the last of these guarantees into this form is simple: $\mathsf{G}(\neg g_1 \vee \neg g_2)$ can be written as a conjunction of an initialisation constraint, $\neg g_1 \vee \neg g_2$, and a basic safety constraint, $\mathsf{G}(\mathbf{true} \rightarrow \mathsf{X}(\neg g_1 \vee \neg g_2))$.

Transforming $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1)$ is a bit more tricky. For storing if after the occurrence of an r_1 request, there is still a g_1 grant to be given, we need some way to transfer this information from one computation cycle to the next one. The system can use additional output signals for this purpose. We add a bit s_1 to represent whether a grant g_1 is still to be given and define its evolution and initialisation by the guarantees $\neg s_1$, $\mathsf{G}(((r_1 \vee s_1) \wedge \neg g_1) \rightarrow \mathsf{X}s_1)$, and $\mathsf{G}((g_1 \vee (\neg s_1 \wedge \neg r_1)) \rightarrow \mathsf{X}\neg s_1)$. We finally add $\mathsf{GF}\neg s_1$ to implement $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1)$. As s_1 will eventually stay **true** forever if and only if a request is issued that is never granted, the property $\mathsf{GF}\neg s_1$ holds on a run of a system that satisfies the new specification if and only if $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1)$ holds.

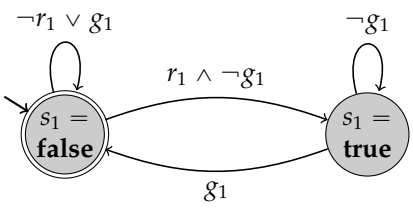

Figure 11.1: Deterministic Büchi automaton for the specification $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1)$ with states labelled by the valution of an auxiliary signal s_1

The same construction can be applied for the guarantee $\mathsf{G}(r_2 \rightarrow \mathsf{F}g_2)$. The overall GR(1) specification then has the input signal set $\mathsf{AP}^I = \{r_1, r_2\}$, the output signal set $\mathsf{AP}^O = \{g_1, g_2, s_1, s_2\}$, and the following guarantees:

1. $\neg g_1 \vee \neg g_2$
2. $\mathsf{G}(\mathbf{true} \rightarrow \mathsf{X}(\neg g_1 \vee \neg g_2))$
3. $\neg s_1$
4. $\mathsf{G}(((r_1 \vee s_1) \wedge \neg g_1) \rightarrow \mathsf{X}\, s_1)$
5. $\mathsf{G}((g_1 \vee (\neg s_1 \wedge \neg r_1)) \rightarrow \mathsf{X}\neg s_1)$
6. $\mathsf{GF}\neg s_1$
7. $\neg s_2$
8. $\mathsf{G}(((r_2 \vee s_2) \wedge \neg g_2) \rightarrow \mathsf{X}\, s_2)$
9. $\mathsf{G}((g_2 \vee (\neg s_2 \wedge \neg r_2)) \rightarrow \mathsf{X}\neg s_2)$
10. $\mathsf{GF}\neg s_2$

★

Note that in this example, one can see why the Mealy-type semantics is useful in GR(1) synthesis: it allows to reason about inputs and outputs of two computation cycles in properties of the form (2) when describing the evolution of the auxiliary signalling bit values. Even more, we only need a single next-time operator in every property in order to do so. This way, we can encode all properties that are representable as deterministic Büchi automata: we allocate sufficiently many state bits, encode the initial state and the transition function of the automaton using initialisation and basic safety constraints, and finally use a liveness property to encode the acceptance of the trace of a system. A deterministic Büchi automaton that represents the LTL formula $\mathsf{G}(r_1 \rightarrow \mathsf{F}g_1)$ is depicted in Figure 11.1, including the information how we mapped the states to variable valuations over s_1 in the constraints 3 to 6 from the property list above, which encode the acceptance of the to-be-synthesised system's trace by the automaton.

The possibility to encode all deterministic Büchi languages in this way motivates the restriction of the property types allowed in the GR(1) synthesis approach to initialisation, basic safety, and basic liveness properties on the specification side. The impact of the approach is however caused by the fact that these property types are easy to deal with on the technical side, too. In fact, the structure of the allowed assumption and guarantee formulas allows us to express all safety assumptions and guarantees as allowed transitions in a game structure whose positions of player 0 are all possible input and output signal valuations $X \subseteq (\mathsf{AP}^I \uplus \mathsf{AP}^O)$. Initialisation properties then select a subset of these, and liveness assumptions and guarantees are encoded into the winning condition of a game that is played on this game structure, which can be represented by the LTL formula

$$(\mathsf{GF}\psi_1 \wedge \ldots \mathsf{GF}\psi_k) \rightarrow (\mathsf{GF}\psi'_1 \wedge \ldots \mathsf{GF}\psi'_l),$$

where all $\psi_1 \ldots \psi'_l$ are LTL formulas that are free of temporal operators.

The resulting game can be transformed to a three-colour parity game, involving a blow-up of a factor of $k \cdot l$ (Bloem et al., 2010a). Such games can be solved in time less than quadratic in the size of the game (Schewe, 2008), and in quadratic time with algorithms that lend themselves to their symbolic implementation (see, e.g., de Alfaro and Faella, 2007). Furthermore, the game structure can be represented using *binary decision diagrams* (*BDDs*, Bryant, 1986) in a very straight-forward way, which fosters the use of symbolic techniques for this synthesis approach in order to speed up computation.

As parity games have memoryless strategies and parity game solving algorithms typically produce such strategies, we immediately obtain an upper bound on the size of the synthesised solutions. Taking this fact together with the low computation time of the parity game solving step, this fact motivated Piterman et al. (2006) to state that the synthesis algorithm requires only $O(N^3)$ time, where N is the state space of the design. Note that their paper does not mention parity games explicitly, but rather presents a solution that uses a specialised algorithm for the type of games they build. However, for the comparison with other synthesis approaches, the parity game viewpoint is beneficial, which is why we adopt it here.

We have already seen in the example above that specifications whose assumptions and guarantees are representable as deterministic Büchi automata can be encoded into the special forms of properties required by the GR(1) approach. Doing this in a way that allows the synthesis procedure to work in an efficient manner is however non-trivial. In fact, for one of the most prominent benchmarks for GR(1) synthesis, the AMBA AHB bus system (ARM Ltd., 1999), the specification was formalised in the GR(1) setting three times (Bloem et al., 2007a,b; Godhal et al., 2011). Each of the two rewritings improved the performance of the synthesis process. This fact shows that a good encoding of the specification in the synthesis process is crucial. To emphasise this additional step, Sohail and Somenzi (2009) coined the term "*pre-synthesis*" for such an encoding operation. Of course one could automate the process, but even in case we already have the best deterministic Büchi automaton (leaving aside what "best" means in this context), encoding an automaton into BDD variables in a way that is most beneficial to efficient symbolic reasoning is a problem on its own (Gosti et al., 2007; Forth and Molitor, 2000).

To conclude the discussion of the generalised reactivity(1) synthesis approach, let us quickly discuss a slight drawback of it. Due to technical reasons, there are specifications for which it incorrectly concludes that a specification is unrealisable although it actually is realisable. Let us explore this idea by an example. Consider the specification $\psi = (\mathsf{G}(\mathbf{true} \rightarrow \mathsf{X}\, r) \wedge \mathsf{GF}\neg r) \rightarrow (\mathsf{G}(\mathbf{true} \rightarrow \mathsf{X}\, g) \wedge \mathsf{G}(\mathbf{true} \rightarrow \mathsf{X}\neg g))$ for $\mathsf{AP}^I = \{r\}$ and $\mathsf{AP}^O = \{g\}$. So we have two assumptions and two guarantees in this case. Note that the conjunction of the assumptions in not satisfiable in this specification: we cannot always get an r from the second computation cycle onwards while obtaining $\neg r$ infinitely often. Thus, this specification is trivially realisable. A generalised reactivity(1) synthesis tool nevertheless produces the result that the specification is unrealisable.

This deviation from the expected result can be explained by the way in which GR(1) synthesis tools work: they construct a game graph whose positions are the possible valuations of the input and output atomic propositions. If in a position p, the environment player can choose a valid next input such that there is no valid output for the system player, then this situation witnesses the unavoidability of the system player to violate the safety guarantees. The synthesis approach then marks p as losing for the system player. However, it actually is not losing: as the environment player has no chance to win in the long run starting from any position in the game, the positions should be marked as being winning for the system player.

Klein and Pnueli (2010) describe the problem in more detail and present a method to fix this problem. It is based on compiling the acceptance condition for the safety assumptions and guarantees into *temporal testers*, and encoding these into the GR(1) specification formulas. In the GR(1) synthesis approach improvements discussed in this thesis however, the correction of this semantic problem comes for free as a by-product.

Chapter 12

Generalised Rabin(1) synthesis

The generalised reactivity(1) synthesis approach (Piterman et al., 2006) has an expressivity problem: we can only apply the approach to specifications that have assumptions and guarantees that are representable as deterministic Büchi automata. Clever pre-synthesis does not help in this respect.[1]

This is an unfortunate situation, as not much is missing to support the assumption and guarantee types needed in practice. Wongpiromsarn et al. (2010a) categorise property types that are helpful for synthesising controllers in the robotics domain. Except for the so-called *stability properties*, all of their property types can be used in the GR(1) synthesis approach after pre-synthesis. Stability properties are LTL formulas of the form $\mathsf{FG}\psi$, where ψ is an LTL formula that is free of temporal operators. These properties can be used to state that a system must eventually behave in a reasonable way, but some finite number of glitches is allowed beforehand. Wongpiromsarn et al. (2010a) describe how such properties can be encoded under the assumption that the system can declare at some point that stability has been reached. In situations in which the system is unable to estimate this point, which is commonly the case if the computation cycle number of the last glitch strictly depends on the input, this solution is not suitable, as the synthesis procedure would incorrectly misclassify the specification to be unrealisable.

Thus, the question whether we can extend GR(1) synthesis to also be able to deal with stability properties suggests itself. To answer this question, we must first fix to what constitutes an extension. As synthesis tools for full LTL are nowadays readily available, and sometimes even optimised for specifications consisting of assumptions and guarantees (Ehlers, 2010a, 2011a), we could always just use such a synthesis tool and consider the problem as having been solved.

The main advantage of GR(1) synthesis over full LTL synthesis procedures is *efficiency*. Efficiency is obtained by having a straight-forward encoding of the synthesis game's transition structure into binary decision diagrams (BDDs), and having a three-colour parity winning condition in the synthesis game. The number of colours does not change with the number of assumptions and guarantees. The constant number of colours greatly simplifies solving the game symbolically, and a fixed-point based game solving algorithm only needs relatively few iterations to analyse the game in this case. Thus, an extension of the idea should have the same properties: a straight-forward symbolic encoding and a parity game with a constant number of colours.

As it turns out, we can accommodate stability properties while retaining the constant number of colours. By applying pre-synthesis, we can then use all assumptions and guarantees with a *Rabin index* of 1. This means that there exists an equivalent deterministic Rabin word automaton for the property with one acceptance pair. This observation justifies the name of the approach, *generalised Rabin(1) synthesis*, abbreviated by *GRabin(1) synthesis*.

In the next section, we discuss the general approach, i.e., we extend the allowed assumption and guarantee types in the GR(1) synthesis approach by stability properties, and show how to perform synthesis in this setting. In Chapter 12.2, we examine the expressivity of the new approach, and deal with efficient pre-synthesis. We will afterwards see that the expressivity of the GRabin(1) synthesis approach cannot be extended significantly without losing its good properties. Then, in Section 12.4, we will discuss an important application of the new approach: the automatic construction of *robust* systems. We conclude with a summary.

12.1 Generalised Rabin(1) synthesis in the direct way

Reconsider the three supported types of assumptions and guarantees of generalised reactivity(1) synthesis described in Chapter 11. In generalised Rabin(1) synthesis, we extend both the allowed assumption

[1] There is of course always the possibility to construct an implementation and then describe it by listing its transitions in the GR(1) specification form, but this defeats the purpose of synthesis.

and guarantee types by *stability properties*, which intuitively say that eventually, nothing bad shall happen any more. Additionally, we also allow liveness and stability properties to relate the current and next state of the system under design (and its environment). So the allowed assumption types in this context are:

(1) ψ_I,

(2) $\mathsf{G}(\psi \rightarrow \mathsf{X}(\psi_I))$,

(3) $\mathsf{GF}(\hat{\psi}_I)$, and

(4) $\mathsf{FG}(\hat{\psi}_I)$,

where ψ is a Boolean formula over $\mathsf{AP}^I \uplus \mathsf{AP}^O$, ψ_I is a Boolean formula over AP^I, and $\hat{\psi}_I$ is an LTL formula for which the only temporal operator allowed in $\hat{\psi}_I$ is X and for which for every occurrence of a next-time operator $\mathsf{X}\phi$, ϕ is a Boolean formula over AP^I.

The allowed guarantee types for GRabin(1) synthesis are:

(1) ψ,

(2) $\mathsf{G}(\psi \rightarrow \mathsf{X}(\psi'))$,

(3) $\mathsf{GF}(\hat{\psi})$, and

(4) $\mathsf{FG}(\hat{\psi})$,

where ψ and ψ' are Boolean formulas over $\mathsf{AP}^I \uplus \mathsf{AP}^O$ and $\hat{\psi}$ is an LTL formula over $\mathsf{AP}^I \uplus \mathsf{AP}^O$ for which the only temporal operator allowed in $\hat{\psi}$ is X and for which for every occurrence of a next-time operator $\mathsf{X}\phi$, ϕ is a Boolean formula over $\mathsf{AP}^I \uplus \mathsf{AP}^O$.

The newly supported constraint type is the one that is numbered by (4). These *stability* properties are sometimes also called *persistence* requirements (Wongpiromsarn et al., 2010b).

In this section, we are concerned with encoding the synthesis problem for such specifications into parity games with a constant number of colours such that precisely the winning strategies for the parity game represent implementations that satisfy the specification.

The presentation below solves the semantic problem in the GR(1) synthesis approach described at the end of Chapter 11 along the way by essentially removing all dead-ends from the game.

Let $\psi = (\bigwedge_{\phi \in A} \phi) \rightarrow (\bigwedge_{\phi \in G} \phi)$ be a GRabin(1) specification for an interface $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$. For simplicity of the following definition, we henceforth define an instance of a GRabin(1) synthesis problem as a tuple $(\mathsf{AP}^I, \mathsf{AP}^O, A_I, A_S, A_L, A_C, G_I, G_S, G_L, G_C)$, where A_I, A_S, A_L, A_C and G_I, G_S, G_L, G_C are sets of assumption and guarantee properties, where we have partitioned the properties according to their type. A_I and G_I are the initialisation properties, A_S and G_S are the safety/succession properties (without the leading G operators), A_L and G_L are liveness properties (without the leading GF operators), and A_C and G_C are the stability properties (without the leading FG operators, the C stands for co-Büchi acceptance). Without loss of generality, we assume that $A_L = \{\phi_1, \ldots, \phi_m\}$ and $G_L = \{\phi'_1, \ldots, \phi'_n\}$, as they need to be numbered in the following construction.

Definition 46. *Let $(\mathsf{AP}^I, \mathsf{AP}^O, A_I, A_S, A_L, A_C, G_I, G_S, G_L, G_C)$ be a GRabin(1) synthesis specification. We define its corresponding five-colour parity game as a tuple $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{init}, c)$ with:*

- $V^0 = (2^{\mathsf{AP}^I} \times 2^{\mathsf{AP}^O} \times \{0, \ldots, m\} \times \{0, \ldots, n\} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}) \cup \{v_{init}\}$
- $V^1 = V^0 \times 2^{\mathsf{AP}^I}$
- *For all $v = (X, Y, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r) \in V^0$ and $X' \subseteq \mathsf{AP}^I$, we have $E^0(v, X') = (X, Y, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r, X')$*
- *For all $v = (X, Y, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r, X') \in V^0$ and $Y' \subseteq \mathsf{AP}^O$, we have $E^1(v, Y') = (X', Y', i', j', f'_a, f'_g, f'_{ca}, g'_{cg}, f'_r)$ with:*
  - *$i' = (i + 1) \bmod m$ if $(X \cup Y)(X' \cup Y') \models \phi_i$ or $i = 0$, otherwise $i' = i$*
  - *$j' = (j + 1) \bmod n$ if $(X \cup Y)(X' \cup Y') \models \phi'_j$ or $j = 0$, otherwise $j' = j$*

  - $f'_a =$ **true** *if* $(X \cup Y)(X') \not\models \phi$ *for some* $\phi \in A_S$*, otherwise* $f'_a = f_a$
  - $f'_g =$ **true** *if* $(X \cup Y)(X' \cup Y') \not\models \phi$ *for some* $\phi \in G_S$*, otherwise* $f'_g = f_g$
  - $f'_{ca} =$ **true** *if* $(X \cup Y)(X') \not\models \phi$ *for some* $\phi \in A_C$
  - $f'_{cg} =$ **true** *if* $(X \cup Y)(X' \cup Y') \not\models \phi$ *for some* $\phi \in G_C$
  - $f'_r =$ **true** *if and only if (at least) one of the following two conditions hold:*
    * $i = 0$ *and* $f_a =$ **false**
    * $f_{cg} =$ **false** *and* $f_r =$ **true**

- *For all* $X \subseteq \mathsf{AP}^I$*, we have* $E^0(v_{init}, X) = (v_{init}, X)$
- *For all* $X \subseteq \mathsf{AP}^I$ *and* $Y \subseteq \mathsf{AP}^O$*, we have* $E^1((v_{init}, X), Y) = (X, Y, 0, 0, f_a, f_g,$ **false**, **false**, **true**$)$ *with* $f_a =$ **false** *if and only if for all* $\phi \in A_I$*, we have* $X \models \phi$*, and* $f_g =$ **false** *if and only if for all* $\phi \in G_I$*, we have* $(X \cup Y) \models \phi$
- *For all* $v = (X, Y, i, j, f_a, f_g, f_r) \in V^0$*, we have that* c *maps* v *to the* ***least*** *value* k *in* $\{0, 1, 2, 3, 4\}$ *such that:*
  - $k = 4$ *if* $f_{ca} =$ **true**
  - $k \geqslant 3$ *if* $f_r =$ **true** *and* $f_{cg} =$ **true**
  - $k \geqslant 2$ *if* $j = 0$ *and* $f_g =$ **false**
  - $k \geqslant 1$ *if* $i = 0$ *and* $f_a =$ **false**
- *For all* $v \in V^1$*,* $c(v) = 0$

Before proving the correctness of the construction, let us discuss its structure. There is no dead-end in the game. For the game to be suitable as a synthesis game for the original specification, we must ensure that precisely the decision sequences that correspond to words that satisfy the specification are winning for player 1 (the system player) in the game.

The game starts in the position v_{init}, in which player 0 can choose the first input. Then, player 1 can choose an output, leading to a transition to a "normal" position $(X, Y, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r)$ of player 0. The flag f_a represents if some safety or initialisation assumption has already been violated along a play in the game so far. It is thus set to **true** along this transition if and only if this was the case in player 0's first move. Likewise, f_g represents if some safety or initialisation guarantee has already been violated.

In the remaining part of the game, player 0 always chooses an input and moves to a position that carries all of the information of the old position, plus additionally the last input. Then, player 1 chooses an output, and a transition is taken in which the last input and output that is stored in the states of V^0 is updated, along with the counters i and j and the flags f_a, f_g, f_{ca}, f_{cg}, and f_r.

The counters i and j are used to cycle through the liveness assumptions and guarantees. This means that at every point in time, there is one liveness assumption and one liveness guarantee that is currently *under observation*. Once we have seen for a liveness property $\mathsf{GF}\phi$ that the last transition satisfied ϕ, we move on to the next property with our pointer. When it hits the maximum, it is reset to 0, and the position that we are then in has a colour of at least 1 (for liveness assumptions) or 2 (for liveness guarantees), unless we already know that some basic safety or initialisation assumption/guarantee has been violated. If all liveness assumptions are fulfilled, then we do not eventually get stuck at some counter value of i, and thus, we visit a position with colour $\geqslant 1$ infinitely often (unless some initialisation or basic safety assumption is violated). The same holds for the liveness guarantees: if they are all fulfilled, then we cycle through a value of 0 for j infinitely often, leading to visiting a position with colour $\geqslant 2$ infinitely often (unless some initialisation of basic safety guarantee is violated). On the other hand, if not all liveness assumptions/not all liveness guarantees are fulfilled, then the i/j counter gets stuck eventually, which prevents visits to positions with colour $\geqslant 1/\geqslant 2$, respectively. So far, this construction is essentially the same as the one for GR(1) games described by Bloem et al. (2010a), but we take safety assumption and guarantee violations into account, and do not assume that such considerations are already built into the game that the construction by Bloem et al. (2010a) starts with.

Note that if we do not have any assumption or guarantee stability properties, then we never visit a position with a colour of 3 or 4 along a play in the game. Thus, we only have a three-colour parity game for such settings. Table 12.1 describes the possible cases for the satisfaction of initialisation/basic safety/liveness assumptions and guarantees and what the maximum colour visited infinitely often in the respective setting is.

		SA	SA LA	LA	
SG		0	1	0	0
SG	LG	2	2	2	2
	LG	0	1	0	0
		0	1	0	0

Table 12.1: Maximal colour occurring infinitely often along plays in games built according to Definition 46 for specifications without stability properties. The colour values depend on whether **all** safety and initialisation assumptions are fulfilled (SA), **all** liveness assumptions are fulfilled (LA), **all** safety and initialisation guarantees are fulfilled (SG), and **all** liveness guarantees are fulfilled (LG). The cases in which the decision sequences corresponding to a play satisfy the specification when read as a word are marked in grey.

To extend this three-colour game encoding to also support stability properties, the main idea is that we can add another two colours for which visiting any of them infinitely often shall witness the violation of a stability property. The colour for stability guarantee violations needs to be odd and > 2, such as colour value 3, to override liveness guarantee satisfactions. On the other hand, stability assumption violations should in turn override stability guarantee violations, and must be even, so we assign to them a colour of 4. As in our game, we need to observe the input and output in two computation cycles to detect the violation of ϕ for some stability assumption or guarantee $\mathsf{FG}\,\phi$, we can only detect this violation along transitions. To mark the positions with colours that represent whether ϕ for some stability assumption or guarantee $\mathsf{FG}\,\phi$ has just been violated, we use the flags f_{ca} and f_{cg} in the game positions.

Table 12.2 describes the properties of the game that we get when applying these ideas. One can see from the table that they do not immediately work - in the cases in which not all stability guarantees are satisfied, but also some safety or liveness assumption is violated, a play is incorrectly classified as being losing for the system player. To fix this problem, we must ensure that liveness and safety guarantees need to be satisfied to visit a position with colour 3 infinitely often. This is precisely where the flag f_r in the position space of the game kicks in. For a position to have colour 3, the flag must be **true**. The flag is set to true if the assumption counter has just cycled through and no safety assumption is yet violated, and stays true until a position with colour 3 has been visited. Whenever this happens, it is reset. Thus, after a safety assumption has been violated, we can only visit a position with colour 3 at most once more, and for visiting colour 3 infinitely often, me must infinitely often cycle though the liveness assumptions. When the flag is taken into account, we obtain the highest-colour table as depicted in Table 12.3.

This one is correct - precisely the decision sequences that satisfy the specification are the ones that induce plays on which the highest colour occurring infinitely often is even. Let us prove this.

Theorem 12. *Let $(\mathsf{AP}^I, \mathsf{AP}^O, A_I, A_S, A_L, A_C, G_I, G_S, G_L, G_C)$ be a GRabin(1) specification, and $\mathcal{G}$ be a game built from the specification according to Definition 46. Precisely the decision sequences that induce winning plays for player 1 (who controls AP^O) satisfy the specification.*

Proof. Let $w = w_0 w_1 \ldots$ be a joint decision sequence of the two players, where for every w_i, we have $w_i = (w_i^I, w_i^O)$. We will show that whenever w satisfies the specification, then the highest colour occurring infinitely along the play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \ldots$ induced by w is even, and if w does not satisfy the specification, then the number is odd. Since w can only either satisfy the specification or not, this suffices for the proof. We sub-divide the two cases into sets of reasons for the satisfaction/non-satisfaction of the specification, and for every case, assume without loss of generality that we are dealing with a word w that is not covered by an earlier case. Note that only positions of player 0 have colours other than 0. Also, there

			CA SA	CA SA LA	CA LA	CA	SA	SA LA	LA	
CG	SG		0	1	0	0	4	4	4	4
CG	SG	LG	2	2	2	2	4	4	4	4
CG		LG	0	1	0	0	4	4	4	4
CG			0	1	0	0	4	4	4	4
	SG		3	3	3	3	4	4	4	4
	SG	LG	3	3	3	3	4	4	4	4
		LG	3	3	3	3	4	4	4	4
			3	3	3	3	4	4	4	4

Table 12.2: Maximal colour occurring infinitely often along plays in games built according to Definition 46, except that we ignore the f_r flag. The values depend on whether **all** safety and initialisation assumptions are fulfilled (SA), **all** liveness assumptions are fulfilled (LA), **all** stability assumptions are fulfilled (CA), **all** safety and initialisation guarantees are fulfilled (SG), **all** liveness guarantees are fulfilled (LG), and **all** stability guarantees are fulfilled (CG). The cases in which the decision sequences corresponding to a play satisfy the specification when read as a word are marked in grey, incorrect classifications are marked by darker grey.

are no dead-ends in the game, so we only need to see which colours occur infinitely often along π. Furthermore, the construction ensures that for all $i > 0$, we have $\pi_i^0 = (w_{i-1}^I, w_{i-1}^O, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r)$ for some values of i, j, f_a, f_g, f_{ca}, f_{cg}, and f_r, so we can read off the respective last moves of the two players from the positions of player 0.

- **Some stability assumption is violated:** Then, there exists some $\phi \in A_C$ such that for infinitely many w_k, we have $(w_k^I \cup w_k^O)(w_{k+1}^I, w_{k+1}^O) \not\models \phi$. The colour of the position π_{k+2}^0 is 4 for every such k (as this triggers flag f_{ca} to be set for these positions), and as 4 is the highest colour, the highest colour occurring infinitely often is 4.
- **Some stability guarantee is not satisfied, but all of the assumptions are satisfied:** Then, i cycles through 0 infinitely often (as all of the liveness assumptions and all safety/initialisation assumptions are fulfilled). This causes f_r to obtain a value of **true** infinitely often, regardless of whether f_r is reset infinitely often or not. As a value of f_r is held until we visit a position with colour 3, f_r has a value of **true** when visiting positions $(X, Y, i, j, f_a, f_g, f_{ca}, f_{cg}, f_r)$ with $f_{cg} =$ **true** infinitely often, which witnesses that for some stability guarantee $\mathsf{FG}\,\phi$, the last two decision sequence elements do not satisfy ϕ. As whenever this happens, we get a colour of 3, and the case occurs infinitely often, we visit colour 3 infinite often.
- **All guarantees are satisfied:** In this case, as all stability properties are satisfied, at some point k, for all $l > k$, we have that for all $\phi \in G_C$, $(w_l^I \cup w_l^O)(w_{l+1}^I \cup w_{l+1}^O) \models \phi$, and thus, a position with colour 3 cannot be visited infinitely often. Furthermore, f_g stays **false**, as we never witness an

			CA SA	CA SA LA	CA LA	CA	SA	SA LA	LA	
CG	SG		0	1	0	0	4	4	4	4
CG	SG	LG	2	2	2	2	4	4	4	4
CG		LG	0	1	0	0	4	4	4	4
CG			0	1	0	0	4	4	4	4
	SG		0	3	0	0	4	4	4	4
	SG	LG	2	3	2	2	4	4	4	4
		LG	0	3	0	0	4	4	4	4
			0	3	0	0	4	4	4	4

Table 12.3: Maximal colour occurring infinitely often along plays in games built according to Definition 46, depending on whether **all** safety and initialisation assumptions are fulfilled (SA), **all** liveness assumptions are fulfilled (LA), **all** stability assumptions are fulfilled (CA), **all** safety and initialisation guarantees are fulfilled (SG), **all** liveness guarantees are fulfilled (LG), and **all** stability guarantees are fulfilled (CG). The cases in which the decision sequences corresponding to a play satisfy the specification when read as a word are marked in grey.

initialisation or safety guarantee violation, and as all liveness guarantees are satisfied, the counter j cannot get stuck at some value, thus cycling through a value of 0 infinitely often. Whenever this happens, the position will have a colour of 2, which will be the highest one occurring infinitely often.

- **Some safety, initialisation, or liveness assumption is violated, but stability assumptions and guarantees are fulfilled:** The fact that all stability properties are fulfilled guarantees that colours 3 and 4 can only be visited finitely often, so only the colours 0, 1, and 2 can occur infinitely often. Colour 1 can only be visited infinitely often if the f_a flag, which witnesses safety and initialisation assumptions failures, remains set to **false**. Furthermore, the i value has to cycle through 0 infinitely often, which can only occur if all liveness assumptions hold. By assumption, one of these two conditions is not fulfilled, and thus we can visit colour 1 only finitely often. Thus, as only the colours 0 and 2 can be visited infinitely often, the highest one occurring infinitely often is even.

- **Some safety, initialisation, or liveness guarantee is not fulfilled, but all safety, initialisation, and liveness assumptions and the stability guarantees are fulfilled:** Again, the fact that all stability properties are fulfilled guarantees that colours 3 and 4 can only be visited finitely often, so only the colours 0, 1, and 2 can occur infinitely often. Furthermore, as visiting colour 2 infinitely often requires, by the same reasoning as for the assumptions in the previous case, initialisation, safety, and liveness guarantees to hold, which is not true in this case, only colours 0 and 1 can be visited infinitely often. As all assumptions are fulfilled, colour 1 will be visited infinitely often as f_a stays set to **false** and counter i cycles through 0 infinitely often.

As we have covered all cases, the claim follows. □

Thus, Definition 46 describes how we can build a *synthesis game* for a GRabin(1) specification in the Mealy-type semantics. The beauty of the construction of Definition 46 is its amenability to applying binary decision diagrams (BDDs, Bryant, 1986; Burch et al., 1992) or other symbolic data structures in a game solving process. For encoding the state space, we only need one bit per signal in $\mathsf{AP}^I \cup \mathsf{AP}^O$, five bits for the flags f_a,f_g, f_{ca},f_{cg}, and f_r, and $\lceil \log_2 |m+1| \rceil + \lceil \log_2 |n+1| \rceil$ bits for encoding the counter values i and j. The position v_{init} can be handled separately. All of these bits have to be allocated twice in order to allow representing E^1 as a BDD.

12.2 Encoding a problem for generalised Rabin(1) synthesis

In the previous section, we have shown how to perform synthesis for specifications in a special shape that allows their direct encoding into games. In many application areas such as, e.g., robotics and reactive planning (Kress-Gazit et al., 2007, 2009; Wongpiromsarn et al., 2010a; Ozay et al., 2011), such specifications occur naturally, as we want to actively reason about the state of the system in these domains, and thus, we have output signals that represent the state anyway.

For specifications that are more complicated, *pre-synthesis* is often needed in order to encode the specification under concern into that format. In this section, we discuss which properties we can handle in the GRabin(1) synthesis approach by performing pre-synthesis, and we discuss how to perform pre-synthesis in a smart way.

For generalised reactivity(1) synthesis, it is well-known that all assumptions and guarantees that are representable as deterministic Büchi automata can be used after pre-synthesis (Morgenstern and Schneider, 2011; Ehlers, 2011b; Bloem et al., 2010a).

Since in GRabin(1) synthesis, we additionally have stability properties at our disposal, the expressivity improves. For example, properties that are representable as deterministic co-Büchi automata are also encodable – by using additional output atomic propositions that encode our position in the automaton, and adding a stability property that says that rejecting states should be visited only finitely often, co-Büchi properties can easily be expressed. Since assumptions and guarantees are treated conjunctively, it immediately follows that we can actually encode all properties that are representable by deterministic Rabin automata with *one* acceptance pair, as such an acceptance condition is a conjunction of a Büchi property and a co-Büchi acceptance condition. Let us formalise this idea and prove the correctness of the construction.

Definition 47. *Let Q be a state set. We need $l = \lceil \log_2 |Q| \rceil$ bits to encode the states into a Boolean vector. We call a function $f : Q \to \mathbb{B}^l$ a* state space encoding function *if for all $q, q' \in Q$, if $q \neq q'$, then $f(q) \neq f(q')$. Let us furthermore define a function m to map a variable valuation to a set of variables $\{x_0, \ldots, x_{l-1}\}$ to a Boolean formula that characterises the valuation. For example, $m((\mathbf{false}, \mathbf{true}, \mathbf{false}, \mathbf{false})) = \neg x_0 \wedge x_1 \wedge \neg x_2 \wedge \neg x_3$.*

A state space encoding allows us to squeeze the state space of an automaton into Boolean variable valuations, as needed to encode a deterministic ω-automaton into the specification form required by GRabin(1) synthesis. There is a trivial canonical encoding for every state set Q: just number the states and assign to every state the binary encoding of its number.

Lemma 15. *Let $\mathcal{A} = (Q, 2^{\mathsf{AP}}, \delta, q_0, \{(E,F)\})$ be a deterministic Rabin automaton with one acceptance pair, and f be some state space encoding for Q over some number l of bits. Without loss of generality, we assume that for all $q \in Q$ and $x \in 2^{\mathsf{AP}}$, we have $|\delta(q,x)| = 1$ (i.e., we have no dead-ends in the automaton). A word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$ is accepted by $\mathcal{A}$ if and only if there exists some attribution $w' = w'_0 w'_1 \ldots \in (2^{\{x_0,\ldots,x_{l-1}\}})^\omega$ such that $(w_0 \cup w'_0)(w_1 \cup w'_1)(w_2 \cup w'_2)\ldots$ satisfies the following LTL constraints:*

(1) $m(f(q_0))$

(2) $\bigwedge_{q \in Q, X \subseteq 2^{\mathsf{AP}}} (m(f(q)) \wedge \bigwedge\{\neg x \mid x \in \mathsf{AP}, x \notin X\} \wedge \bigwedge\{x \mid x \in \mathsf{AP}, x \in X\}) \rightarrow \mathsf{X}(m(f(\delta(q,x))))$

(3) $\mathsf{GF}\left(\bigvee_{q \in F} m(f(q))\right)$

(4) $\mathsf{FG}\left(\bigwedge_{q \in E} \neg m(f(q))\right)$

Furthermore, there is only precisely one such attribution w', and it can be computed eagerly, i.e., for every $i \in \mathbb{N}$, we only need w_i and w'_i to compute w'_{i+1}.

Proof. Let $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$ be a word and $\pi = \pi_0 \pi_1 \ldots$ be the run of $\mathcal{A}$ over w. Then $w' = w'_0 w'_1 \ldots \in (2^{\{x_0,\ldots,x_{l-1}\}})^\omega$ has the property that for all $i \in \mathbb{N}$, we have that $w'_i = f(\pi_i)$. This can be proven by induction on i. For $i = 0$, the property is true by constraint (1). For $i > 0$, the properties of the type (2) ensure this fact. Since there is always only one valuation for w'_i that satisfies the properties of that type, w' can be computed step-by-step while observing w.

As w' has to resemble the run of $\mathcal{A}$ for constraints (1) and (2) to be satisfied, we can add stability and liveness constraints to encode the acceptance of the automaton. The constraints of type (3) and (4) above do precisely this. □

Note that in practice, for automata with a number of states that is not 2^k for some $k \in \mathbb{N}$, up to $\frac{1}{2}$ of the bit valuations for $\{x_0, \ldots, x_{l+1}\}$ (minus one) are actually not needed. We can then extend the idea above to let a state space encoding f map a state to multiple different variable valuations that can all be used for encoding a state, i.e., we have $f : Q \to 2^{\mathbb{B}^l}$ with the constraint that for all $q, q' \in Q$, if $q \neq q'$, then $f(q) \cap f'(q) = \varnothing$. The function m then maps a set of variable valuations to a formula that represents the *disjunction* of the possible valuations. The lemma above still holds after this modification, with the only difference that in the proof, variable valuations that refer to the same state are treated as being equivalent.

Finding a good state space encoding function f that leads to an efficient synthesis process is a problem on its own. Fortunately, we can apply research on computing small circuits from finite-state machines here. The SIS logic synthesis toolset (Sentovich et al., 1992), for example, comes with a tool called NOVA, which computes small circuits that implement explicitly given finite-state machines. Apart from the circuit, it also describes the state space encoding that was used when constructing it. Experiments on state space encoding that were conducted by the author of this thesis in the scope of the timed synthesis and verification tool SYNTHIA (Peter et al., 2011) suggest that the encodings produced by NOVA are well-suited for BDD-based games solving.

By the construction of Lemma 15, we can replace a deterministic Rabin automaton with one acceptance pair that represents an assumption or guarantee property in a specification by a set of LTL formulas that encode the meaning of the automaton. For this process to work in the scope of synthesis, we need to ensure that the additional attribution bits belong to the right player in the game.

If we are encoding a guarantee, then the *system* player has an interest to always assign the only correct values to the attribution bits, as otherwise, some guarantee is immediately violated. This way, a synthesised system outputs the evolution of the run through the Rabin automaton as a side-effect. As the output can however easily be ignored, this is no drawback.

If we are encoding an assumption, things are a bit different. Attributing the additional bits to the system player would not work, as then, the system player has an incentive to play an illegal move and make the assumptions unsatisfied. Thus, we need to attribute the additional signals as inputs to the environment player and add the pre-synthesised properties to the set of assumptions. For deciding the realisability of a specification, this is sufficient - the case that the additional input always represents the correct state of the Rabin automaton is the worst case here, and the only one that needs to be considered - in all other cases, the system player wins. Thus, the realisability/unrealisability result will be correct. However, to obtain an implementation in case of realisability, we need to modify the Mealy machine that implements a winning strategy in the game and take its product with a second process that emulates the Rabin automaton, and that is thus able to always feed some valid input to the additional input variables. Adding the additional process is not difficult. Note that for the original generalised reactivity(1) synthesis approach, this is actually not necessary, as the semantic problem (Klein and Pnueli, 2010) of GR(1) synthesis that we discussed in Chapter 11 allows us to attribute the additional bits and the properties for their evolution to the guarantee player: as this player loses **immediately** when producing incorrect output, she cannot make use of the possibility to simulate transitions in the Rabin automaton incorrectly.

12.2.1 Obtaining minimal Rabin(1) automata

To encode a deterministic Rabin automaton with one acceptance pair as an assumption or guarantee in GRabin(1) synthesis, we must obviously first have it available. Apart from specifying it directly, since

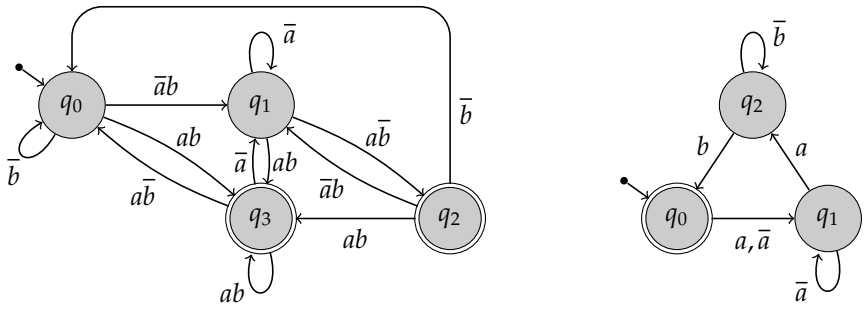

Figure 12.1: A non-minimal DBW (left) and a minimal DBW (right) for the LTL formula $(\mathsf{GF}\, a) \wedge (\mathsf{GF}\, b)$ over the atomic proposition set $\mathsf{AP} = \{a, b\}$. Both DBW have a total transition function. Accepting states are doubly-circled.

the seminal works by Safra (1988, 1989), it is known how to translate an LTL formula into a deterministic Rabin automaton in a complexity-theoretically optimal way. Furthermore, nowadays, suitable tools like ltl2dtar (Klein and Baier, 2006) for this task area are readily available at our disposal.

However, Safra's construction does not guarantee to produce Rabin automata with a minimal number of acceptance pairs, even for specifications for which one pair would suffice, and so does ltl2dstar. Thus, additional steps need to be taken to obtain a one-pair Rabin automaton whenever possible. Krishnan et al. (1995) provided a procedure to minimise the Rabin index in Rabin automata, i.e., the number of acceptance pairs. The procedure can increase the number of states in a Rabin automaton: if we start with n states and h pairs, we obtain an automaton with $h \cdot n$ states if there exists an equivalent one-pair deterministic Rabin automaton. If there exists none, the construction allows us to see this in polynomial time.

For GR(1) synthesis, where we would construct a deterministic Büchi automaton from a given property in order to apply pre-synthesis to it, the situation is actually a bit simpler: Kupferman et al. (2006a) proved that deterministic Rabin automata are *Büchi-type*. This means that there exists a deterministic Büchi automaton whose language is equivalent to the one of a given deterministic Rabin automaton if and only if there exists one with the same transition structure. For finding the Büchi automaton, we can simply search for loops in the Rabin automaton that lead to non-acceptance if we eventually cycle on the loop forever, declare the states that are not contained in any of these loops to be accepting, and check if the language of the computed Büchi automaton is the same. If the language of the Rabin automaton is Büchi-recognisable, then they are the same. All these operations can be performed in polynomial time. Note that the construction by Krishnan et al. (1995), if restricted to the Büchi case, performs precisely this check.

The algorithm by Krishnan et al. (1995) for minimising the Rabin index of an automaton however only minimises the Rabin index, and not the number of states of the automaton. Looking at the complexities of these problems, this is not surprising. If we are only interested in Rabin automata with one acceptance pair, then their algorithm runs in polynomial time. Minimising the number of states (which we will simply call *minimising an automaton* henceforth) is however NP-hard, even if we are only concerned with deterministic Büchi automata, as proven by Schewe (2010). Nevertheless, as the minimisation problem for Rabin(1) automata is contained in the complexity class NP, but synthesis, for which we want to use our deterministic Rabin automata, has an even higher complexity, it is worthwhile to minimise the Rabin(1) automata before using them in synthesis.

It can already be seen from the complexities that minimising deterministic Rabin(1) automata is different from minimising deterministic automata over finite words: while for the latter, there exists a suitable polynomial-time algorithm, which is based on merging states with the same language, the same idea cannot be exploited for Büchi automata. The left part of Figure 12.1 shows a one-pair Rabin automaton (that is even a Büchi automaton) over an alphabet $\Sigma = 2^{\{a,b\}}$ that is equivalent to the LTL formula $(\mathsf{GF}a) \wedge (\mathsf{GF}b)$.This deterministic Büchi word automaton (*DBW*) has four states, whereas the smallest deterministic Büchi or one-pair Rabin automata that are equivalent to this formula only have three states. One such DBW is depicted in the right part of Figure 12.1. It is by no means obvious how to *restructure* the left automaton in order to obtain such a smaller one.

Even more, all states have the same language, so this example shows that we cannot only rely on

language equivalence for minimising deterministic Rabin(1) automata. We thus propose a different approach here. Assume that some n-state automaton $\mathcal{A}' = (Q', \Sigma, \delta', q_0', (E', F'))$ is given. We can assume without loss of generality that $\mathcal{A}'$ has no dead-ends (i.e., for all $q \in Q'$ and $x \in \Sigma$, we have $|\delta(q', x)| = 1$), as otherwise we can simply add a state with self-loops for all $x \in \Sigma$ such that the state is not in F' and $\delta(q', x)$ maps to the state whenever there is no other transition. We use a SAT solver to consider all possible $n-1$-state *candidate automata* $\mathcal{A} = (Q, \Sigma, \delta, q_0, (E, F))$ and encode the equivalence check of the languages of $\mathcal{A}$ and $\mathcal{A}'$ in clausal form. While such a check is PSPACE-complete for non-deterministic Büchi automata, it can be performed in polynomial time for deterministic one-pair Rabin automata and can also be efficiently encoded into a SAT instance. By repeatedly applying this reduction technique until the resulting SAT instance becomes unsatisfiable, we obtain an automaton of minimal size.

For encoding the equivalence check, we use the observation that we can deduce from the *product* of $\mathcal{A}$ and $\mathcal{A}'$ if the two deterministic Rabin automata have the same language. More precisely, we build the graph $G = \langle V, \hat{E} \rangle$ with the set of vertices $V = Q \times Q'$ and edges $\hat{E} = \{((q_1, q_1'), (q_2, q_2')) \mid \exists s \in \Sigma : \delta(q_1, s) = q_2 \wedge \delta'(q_1', s) = q_2'\}$. If there is some loop $(v_0, v_0')(v_1, v_1') \dots (v_k, v_k')(v_0, v_0')$ in $\langle V, \hat{E} \rangle$ with (v_0, v_0') being reachable from (q_0, q_0') such that the projection of the loop onto $\mathcal{A}$ is accepting, but it is not for $\mathcal{A}'$, then there exists some $w \in \mathcal{L}(\mathcal{A})$ such that $w \notin \mathcal{L}(\mathcal{A}')$, so $\mathcal{A}$ and $\mathcal{A}'$ are inequivalent. Dually, if the projection of the loop onto $\mathcal{A}'$ is accepting, but it is not for $\mathcal{A}$, there exists a word $w \notin \mathcal{L}(\mathcal{A})$ such that $w \in \mathcal{L}(\mathcal{A}')$. If no such loops can be found, $\mathcal{A}$ and $\mathcal{A}'$ are equivalent.

Using the observation stated above, we can build a SAT problem instance for solving the deterministic Rabin(1) automaton state reduction problem. We use the following set of variables:

$$\{\langle q \rangle_E, \langle q \rangle_F \mid q \in Q\} \cup \{\langle q_1, s, q_2 \rangle_\delta \mid q_1, q_2 \in Q, s \in \Sigma\} \cup \{\langle q, q' \rangle_G \mid q \in Q, q' \in Q'\}$$
$$\cup \{\langle q_1, q_1', q_2, q_2' \rangle_{(b_1,b_2,b_3,b_4)} \mid q_1, q_2 \in Q, q_1', q_2' \in Q', b_1, b_2, b_3, b_4 \in \mathbb{B}, \neg b_1 \vee \neg b_3\}$$

The SAT clauses are defined as follows:

$$\bigwedge_{q_1 \in Q, s \in \Sigma} \bigvee_{q_2 \in Q} \langle q_1, s, q_2 \rangle_\delta \tag{12.1}$$

$$\wedge \bigwedge_{q_1, q_2 \in Q, q' \in Q', s \in \Sigma} \langle q_1, q' \rangle_G \wedge \langle q_1, s, q_2 \rangle_\delta \rightarrow \langle q_2, \delta'(q', s) \rangle_G \tag{12.2}$$

$$\wedge \langle q_0, q_0' \rangle_G \wedge \bigwedge_{q \in Q, q' \in Q'} \left(\langle q, q' \rangle_G \Rightarrow \langle q, q', q, q' \rangle_{(\mathbf{false},\mathbf{false},\mathbf{false},\mathbf{false})} \right) \tag{12.3}$$

$$\wedge \bigwedge_{\substack{q_1, q_2, q_3 \in Q, q_1', q_2' \in Q', s \in \Sigma, b_1, \dots, \\ b_4 \in \mathbb{B}, \neg b_1 \vee (\neg b_3 \wedge \delta(q_2', x) \notin E)}} \begin{array}{c} \langle q_1, q_1', q_2, q_2' \rangle_{(b_1,b_2,b_3,b_4)} \wedge \langle q_2, s, q_3 \rangle_\delta \wedge \neg \langle q_3 \rangle_E \wedge \neg \langle q_3 \rangle_F \\ \rightarrow \quad \langle q_1, q_1', q_3, \delta'(q_2', s) \rangle_{(b_1,b_2,b_3 \vee \delta'(q_2', x) \in E, b_4 \vee \delta'(q_2', x) \in F)} \end{array} \tag{12.4}$$

$$\wedge \bigwedge_{\substack{q_1, q_2, q_3 \in Q, q_1', q_2' \in Q', s \in \Sigma, \\ b_1, \dots, b_4 \in \mathbb{B}, \neg b_3, \delta'(q_2', x) \notin E}} \rightarrow \begin{array}{c} \langle q_1, q_1', q_2, q_2' \rangle_{(b_1,b_2,b_3,b_4)} \wedge \langle q_2, s, q_3 \rangle_\delta \wedge \langle q_3 \rangle_E \wedge \neg \langle q_3 \rangle_F \\ \langle q_1, q_1', q_3, \delta'(q_2', s) \rangle_{(\mathbf{true},b_2,\mathbf{false},b_4 \vee \delta'(q_2', x) \in F)} \end{array} \tag{12.5}$$

$$\wedge \bigwedge_{\substack{q_1, q_2, q_3 \in Q, q_1', q_2' \in Q', s \in \Sigma, \\ b_1, \dots, b_4 \in \mathbb{B}, \neg b_1 \vee (\neg b_3 \wedge \delta'(q_2', x) \notin E)}} \rightarrow \begin{array}{c} \langle q_1, q_1', q_2, q_2' \rangle_{(b_1,b_2,b_3,b_4)} \wedge \langle q_2, s, q_3 \rangle_\delta \wedge \neg \langle q_3 \rangle_E \wedge \langle q_3 \rangle_F \\ \langle q_1, q_1', q_3, \delta'(q_2', s) \rangle_{(b_1,\mathbf{true},b_3 \vee \delta'(q_2', x) \in E, b_4 \vee \delta'(q_2', x) \in F)} \end{array} \tag{12.6}$$

$$\wedge \bigwedge_{\substack{q_1, q_2, q_3 \in Q, q_1', q_2' \in Q', s \in \Sigma, \\ b_1, \dots, b_4 \in \mathbb{B}, \neg b_3, \delta'(q_2', x) \notin E}} \rightarrow \begin{array}{c} \langle q_1, q_1', q_2, q_2' \rangle_{(b_1,b_2,b_3,b_4)} \wedge \langle q_2, s, q_3 \rangle_\delta \wedge \langle q_3 \rangle_E \wedge \langle q_3 \rangle_F \\ \langle q_1, q_1', q_3, \delta'(q_2', s) \rangle_{(\mathbf{true},\mathbf{true},\mathbf{false},b_4 \vee \delta'(q_2', x) \in F)} \end{array} \tag{12.7}$$

$$\wedge \bigwedge_{q_1 \in Q, q_1' \in Q' b_3, b_4 \in \mathbb{B}, b_3 \vee \neg b_4} \neg \langle q_1, q_1', q_1, q_1' \rangle_{(\mathbf{false},\mathbf{true},b_3,b_4)} \tag{12.8}$$

$$\wedge \bigwedge_{q_1 \in Q, q_1' \in Q' b_1, b_2 \in \mathbb{B}, b_1 \vee \neg b_2} \neg \langle q_1, q_1', q_1, q_1' \rangle_{(b_1,b_2,\mathbf{false},\mathbf{true})} \tag{12.9}$$

$$\wedge \bigwedge_{q_1 \in Q, q_1' \in Q', b_1, b_2 \in \mathbb{B}} \langle q_1, q_1', q_1, q_1' \rangle_{(b_1,b_2,\mathbf{false},\mathbf{true})} \rightarrow \neg \langle q_1 \rangle_E \tag{12.10}$$

Note that all clauses are in conjunctive normal form (CNF) after substituting the $\rightarrow$ operator using its definition ($\phi \rightarrow \phi' = \neg\phi \vee \phi'$ for all sub-formulas ϕ and ϕ'), and are thus immediately usable with a

SAT solver. The variables $\langle\cdot\rangle_E$ and $\langle\cdot\rangle_F$ represent whether a state is in E or F for the Rabin acceptance pair (E, F) of the automaton to compute. The transition function of the candidate automaton is defined in the variables $\langle\cdot\rangle_\delta$. The clauses (12.1) make sure that the transition function assigns a successor for every state/character combination. The clauses (12.3) and (12.2) build our reachability graph: the combination of the initial states in the new candidate automaton and the original reference automaton is always reachable by definition, and the succession clauses (12.2) ensure that the other reachable state combinations in the product have their respective variable set to **true**.

Using our reachability graph, we search for cycles in the product that witness the incorrectness of the candidate automaton. For this, we search from every vertex (q, q') in the product of the two automata what other vertices in the product are reachable, and whether states in E, F, E' and/or F' have been visited along the way. This is reflected in the indices b_1, b_2, b_3, and b_4 of the variables $\langle\cdot\rangle_{(b_1,b_2,b_3,b_4)}$, respectively. We however do not need to store product nodes for which E and E' have been visited since the start of the loop, as a loop starting in this way cannot witness the inequality of the languages. Formulas (12.4) to (12.7) capture the possible cases along this search for loops. Formula (12.8) then states that no reachable loop in the product shall be found along which the candidate automaton accepts, but the reference automaton does not, and Formula (12.9) considers the converse case.

Finally, Formula (12.10) denotes a hint to the SAT solver and is strictly optional. If we find some reachable loop in the product of the reference automaton and the candidate automaton such that the reference automaton accepts when taking the loop infinitely often, then we know that all of the candidate automaton states along the loop must not be in the set E of its acceptance pair (E, F), as otherwise, the candidate automaton cannot accept along *any* loop from these states. Thus, the state in which the loop starts must not be in E in particular.

Note that there are no clauses enforcing that not too many variables in $\langle\cdot\rangle_G$, $\langle\cdot\rangle_\delta$, or $\langle\cdot\rangle_{(b_1,b_2,b_3,b_4)}$ are set to **true**. This is not necessary, as this only makes finding a satisfying assignment harder, but never results in false-positives.

For speeding up the SAT solving process, symmetry breaking clauses (see, e.g., Sakallah, 2009) can also be added. For simplicity, let us only break symmetry partially. In particular, for $Q = \{q_1, \ldots, q_{|Q|}\}$ and $\Sigma = \{s_1, \ldots, s_{|\Sigma|}\}$, we add the following conjuncts that encode some relaxed form of lexicographical minimality of the candidate automaton obtained over the automata whose graphs are isomorphic to the candidate solution:

$$\bigwedge_{1 \leqslant i < |Q|,\, i+1 < j \leqslant |Q|,\, (i-1)\cdot|Q|+j+2 \leqslant k \leqslant |\Sigma|} \neg\langle q_i, s_k, q_j\rangle_\delta$$

Note that a simplified version of the overall construction for the case that the input automaton is representable as a deterministic Büchi automaton has been given by Ehlers (2010b).

12.3 Complexity considerations

We have introduced generalised Rabin(1) synthesis as a fast synthesis approach with sufficient expressivity for most practical applications. Apart from the speed of such an approach in practice, one of the driving ideas was its lower complexity: although the game that we build in the approach has a number of positions that is exponential in the number of atomic propositions, solving the game can be performed in time polynomial in the size of the game. This shows that we are only dealing with a problem in the complexity class EXPTIME. Yet, it remains to show that the algorithm we presented is optimal for the problem. We do so here in three ways. First of all, let us show that the problem is also EXPTIME-hard. Second, as any synthesis approach that reduces the synthesis problem to game solving with exponentially many positions but a constant number of colours is in EXPTIME, it is also important to show that we cannot build synthesis games for our specifications with less than 5 colours. Finally, it is important to analyse whether we can actually continue the line of work by allowing even more assumption and guarantee types without exceeding a constant number of colours.

12.3.1 EXPTIME-completeness of GR(1) synthesis

Let us start with the proof of EXPTIME-hardness of the synthesis problem. The claim already holds for generalised reactivity(1) synthesis, so we do not even need the extension to GRabin(1) synthesis. The main idea of the proof of the following theorem is to reduce the acceptance of an alternating polynomially space-bounded Turing machine to the realisability problem of a generalised reactivity(1)

specification. For a different fragment of LTL, this approach to showing EXPTIME-hardness for the synthesis problem has been followed earlier by Alur and La Torre (2004, Theorem 5.6). The fragment they consider however allows to nest the next-time and finally operators of LTL, which is not allowed in GR(1) specifications. Thus, we follow an alternative route for this reduction in the theorem below.

Theorem 13. *Realisability checking for generalised reactivity(1) specifications is EXPTIME-hard.*

Proof. We reduce the acceptance of an alternating Turing machine with a polynomially bounded tape for a given word onto realisability checking of a GR(1) specification. Without loss of generality, we can assume that the polynomial $p(n)$ that describes the maximal memory needed by the Turing machine is known and easily computable (in polynomial time), and the Turing machine has been modified such that it never moves the tape head left of the tape position on which the first input symbol is put, or right of $p(n)$. This modification can only blow-up the number of states of the Turing machine and the memory needed by it by a polynomial. Furthermore, we can assume that once an accepting state of the Turing machine has been reached, it is never left again. Modifying the machine accordingly is trivial.

We can then encode the run of the Turing machine into a GR(1) specification: we first of all take sufficiently many output bits to encode the state of the machine and also allocate enough output bits to describe the position of the tape head. Then, we add enough bits to output all $p(n)$ tape cell contents in parallel. We take only one input bit. As constraints, we encode the following guarantees (no assumptions are needed):

- Initially, the tape content is the input word to the Turing machine, followed by the designated empty character.
- In every computation step, for all tape positions that are at least one position away from the tape head, the cell content in the next computation cycle is the same as the current tape cell content.
- In every computation step, the combination of the next tape content on the current position, the next position, and the next state, is one of the allowed successor configurations of the current state and the current tape content. Without loss of generality, we can assume that there are always precisely two possible successor state/tape cell content combinations. In universal states, the choice of which of these two combinations is taken depends on the input bit. In existential states, any successor combination is fine.
- The state of the Turing machine is accepting infinitely often.

The number of guarantees in the final GR(1) specification is linear in $p(n)$.

Assume that the alternating Turing machine accepts the input word. Then, there exists some acceptance tree, i.e., some configuration graph such that along every path in the tree, the Turing machine accepts. Since our construction ensures that we can then always take the configurations along this tree, the system player has a winning strategy that ensures that eventually, the system outputs some accepting state. As it is then stuck in this state, the fourth constraint is also satisfied.

On the other hand, assume that there exists some winning strategy. Then, as the synthesised solution is forced to output valid tape configurations, all transitions have to represent valid evolutions of the Turing machine's configurations, and take the input into account for universal branching, we can deduce from the fact that the implementation guarantees that eventually some accepting state of the Turing machine is visited that the Turing machine accepts the word. □

As the realisability problem for GR(1) specifications is EXPTIME-hard, and we have an EXPTIME-algorithm for generalised Rabin(1) synthesis (including the computation of an implementation), EXPTIME-completeness of GRabin(1) realisability checking follows.

12.3.2 We really need five colours for GRabin(1) synthesis

We have reduced GRabin(1) synthesis to parity game solving with 5 colours. The number 5 seems rather arbitrary, so the question is natural whether we can actually do the same with fewer colours.

It turns out that we cannot. Consider the case that we want to synthesise a system from a specification that is representable as a deterministic parity automaton with the five colours $\{0, 1, 2, 3, 4\}$. We can encode the realisability and synthesis problems for such a specification into GRabin(1) synthesis by performing pre-synthesis to map the transitions of the automaton onto safety and initialisation assumptions, and

then using the following overall specification, where $\phi_{\sim c}$ denotes the set of automaton states whose colours are $\sim c$ for some $c \in \mathbb{N}$ and $\sim \in \{\leqslant, \geqslant\}$:

$$\left(\mathsf{GF}\phi_{\geqslant 1} \wedge \mathsf{FG}\phi_{\leqslant 3} \wedge \psi_{\text{automaton simulated correctly}}\right) \rightarrow \left(\mathsf{GF}\phi_{\geqslant 2} \wedge \mathsf{FG}\phi_{\leqslant 2}\right) \tag{12.11}$$

We thus ask the environment player to provide the correct automaton simulation on some extra input signals of sufficient cardinality to encode the automaton. Making a mistake in providing the correct input along these extra bits is the worst case for the environment player, so we can assume that the stream is given correctly. Using the specification above, the system player can then win if and only if there exists a strategy for her to satisfy the specification imposed by the original deterministic parity automaton. This can be seen from the fact that the liveness and stability assumptions and guarantees encode precisely the acceptance of the parity automaton: if it is not colour 0 that is the highest colour visited infinitely often ($\mathsf{GF}\phi_{\geqslant 1}$), and it is not colour 4 ($\mathsf{FG}\phi_{\leqslant 3}$), then it has to be colour 2 ($\mathsf{GF}\phi_{\geqslant 2} \wedge \mathsf{FG}\phi_{\leqslant 2}$). Furthermore, a strategy for the GRabin(1) game built is also a correct implementation that satisfies the original deterministic parity automaton, provided that we attach another process that feeds the respective current automaton state to the synthesised implementation. As the parity hierarchy is strict (i.e., for every $i \in \mathbb{N}$, there exists a language that can be represented as a deterministic i-colour parity automaton, but not as a deterministic $(i-1)$-colour parity automaton, Niwinski and Walukiewicz, 1998), the fact that we need five colours for GRabin(1) synthesis, but could also encode languages that need five colours as synthesis specifications, shows us that in the colour-counting sense, the construction proposed in this chapter is optimal.

Note that strictly speaking, the specification in equation (12.11) is not in GRabin(1) form as the safety assumption (set) that the automaton is simulated correctly relates both input and output in two computation cycles of the system to be synthesised, while we defined that in the successor computation cycle, only the input may be used in a safety assumption. To circumvent that problem, we can add additional input bits that the input player can use for forwarding the input and output in one computation cycle to the next one. By adding some more assumptions, this usage of the additional signals can be enforced, and the five-colour parity automaton can then be simulated with a delay of one. While this procedure is very inefficient, this does not matter at this point, as the aim here was only to establish the optimality in the number-of-colours sense, and we have nevertheless encoded a five-colour parity automaton as specification in GRabin(1) synthesis.

12.3.3 On extending GRabin(1) synthesis

The generalised Rabin(1) synthesis approach presented in this thesis is capable of handling all assumptions and guarantees that have a Rabin index of one. A natural question to ask at this point is whether the approach can be extended in order to also be able to handle specifications with conjuncts of a higher Rabin index without losing its good properties. These are:

- the fact that the state space of the game is the product of all atomic proposition valuations plus some control structure that does not blow-up the game by more than a polynomial.
- the constant number of colours, which allows the application of efficient symbolic parity game solving algorithms.

Unfortunately, the approach cannot be extended significantly while retaining these advantages. To see this, consider Streett game solving, which is known to be co-NP-complete (see, e.g., Grädel et al., 2002). If we were able to accommodate one-pair Streett automata (which are a special case of two-pair Rabin automata) as guarantees in the synthesis approach presented, we could decompose a Streett automaton with n acceptance pairs (for some $n \in \mathbb{N}$) into n one-pair Streett automata, each having the transition structure of the original Streett automaton, take these as guarantees and use no assumptions. The specification is then realisable if and only if it is realisable for the original Streett automaton. Since however, the individual one-pair Streett automata have the same transition structure and thus transition in a synchronised manner, if we were able to build a parity game having the properties stated above, the parity game would only have a number of positions that is polynomial in the size of the original Streett automaton and thus, due to the constant number of colours, the realisability problem for the original Streett automaton would be solvable in polynomial time. So the existence of a similar algorithm for *generalised Streett(1) synthesis* or *generalised Rabin(2) synthesis* would imply P=NP.

12.4 Synthesising robust systems

Assume that we have a specification of the form $(a_1 \wedge a_2 \wedge \ldots \wedge a_{n_a}) \rightarrow (g_1 \wedge g_2 \wedge \ldots \wedge g_{n_g})$, where all assumptions and guarantees are either initialisation, basic safety, basic liveness or stability properties. During the run of a system satisfying ψ, the assumptions may be violated temporarily. A common criterion for the robustness of a system is that it must at some point return to *normal operation mode* after such a temporary assumption violation (Bloem et al., 2009; Arora and Gouda, 1993). In the scope of synthesis, implementing such a *convergence criterion* (Arora and Gouda, 1993) requires fixing a definition of temporary assumption violations. Taking a specification of the form needed by generalised reactivity(1) synthesis, as described in Chapter 11, only a violation of the initialisation or basic safety assumptions can be detected during the run of the system. Moreover, only the basic safety properties can be violated temporarily as an initialisation property is only evaluated at the start of a system's run. Thus we define:

Definition 48. *Given a word $w = w_0 w_1 \ldots \in (2^{\mathsf{AP}})^\omega$ and an LTL formula $\psi = \mathsf{G}\,\phi$, we say that position $i \in \mathbb{N}$ in the word* witnesses the non-satisfaction *of ψ if there exists some $j < i$ such that for no $w' \in (2^{\mathsf{AP}})^\omega$, $w_j \ldots w_i w' \models \phi$ but there exists some $w' \in (2^{\mathsf{AP}})^\omega$ such that $w_j \ldots w_{i-1} w' \models \phi$. Furthermore, given a specification of the form $(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \rightarrow (g_1 \wedge g_2 \wedge \ldots \wedge g_n)$, where all assumptions and guarantees are initialisation, basic safety, basic liveness or stability properties, we say that the assumptions/guarantees are* temporarily violated *on a word w at position i if position i witnesses the non-satisfaction of some basic safety assumption/guarantee in the specification, respectively.*

Arora and Gouda (1993) defined *convergence* for the case of safety specifications. We extend the definition to the liveness and persistence cases:

Definition 49. *Given a specification of the form $(a_1 \wedge a_2 \wedge \ldots \wedge a_m) \rightarrow (g_1 \wedge g_2 \wedge \ldots \wedge g_n)$, where all assumptions and guarantees are initialisation, basic safety, basic liveness and persistence properties, we say that a system converges if the following conditions hold for all words in the language of the system:*

- *there exists a bound on the number of temporary basic safety guarantee violations between any two temporary basic safety assumption violations and after the last temporary basic safety assumption violation, and*
- *If w is a word in the language of the system satisfying the initialisation assumptions and for some $j \in \mathbb{N}$, w^j satisfies all non-initialisation assumptions, then w satisfies the initialisation guarantees and for some $j' \geqslant j$, $w^{j'}$ satisfies all non-initialisation guarantees.*

In this definition, there is no requirement that a converging system also performs some progress on satisfying its liveness (and persistence) properties in between two temporary assumption violations if they are sufficiently sparse, which in practice a robust system should surely do. Nevertheless, the definition is still useful. The reason is that all synthesis procedures used nowadays produce finite-state solutions. Thus, if temporary assumption violations stop occurring for a couple of computation cycles and at the same time, some progress is made with respect to fulfilling the liveness assumptions, the system has, after a finite period of time, also continue to work towards fulfilling the liveness and stability guarantees. If this was not the case, we could find a loop in the finite-state machine description of the system that witnesses non-convergence when taking it (and only it) infinitely often, which would be a contradiction. Given that our synthesis procedure only produces finite-state solutions (as parity game solving algorithms typically do), it is thus enough to require that after the *last* temporary assumption violation, the system converges in order to ensure that the system converges in general. We can easily express convergence after the last temporary assumption violation in LTL by prefixing the guarantees and the basic safety assumptions in the specification with the F (finally) operator of LTL.

Definition 50. *Given a specification of the form $\psi = (a_1 \wedge a_2 \wedge \ldots \wedge a_m) \rightarrow (g_1 \wedge g_2 \wedge \ldots \wedge g_n)$, where all assumptions and guarantees are initialisation, basic safety, basic liveness or persistence properties, $a_1, \ldots, a_s$ are precisely the initialisation assumptions, and $g_1, \ldots, g_{s'}$ are precisely the initialization guarantees, we define the* ruggedised version of *ψ to be:*

$$\psi' = (a_1 \wedge \ldots \wedge a_s \wedge \mathsf{F}(a_{s+1}) \wedge \ldots \wedge \mathsf{F}(a_m)) \rightarrow (g_1 \wedge \ldots \wedge g_{s'} \wedge \mathsf{F}(g_{s'+1}) \wedge \ldots \wedge \mathsf{F}(g_n))$$

Note that when taking a generalised Rabin(1) specification consisting only of initialisation, basic safety, basic liveness and persistence properties, ruggedising it does not change its membership in

this class, as basic safety properties are converted to persistence properties, basic liveness properties remain untouched (as $\mathsf{FGF}(\phi)$ is equivalent to $\mathsf{GF}(\phi)$ for all LTL formulas ϕ) and likewise, persistence properties are not altered. This property does not hold for generalised reactivity(1) specifications, as the ruggedisation process converts basic safety properties to persistence properties.

Thus, we can solve the robust synthesis problem, where converging systems that satisfy a given specification are to be found, by ruggedising the specification and using the generalised Rabin(1) synthesis approach.

So far, we have required all assumption and guarantee conjuncts to be initialisation, basic safety, basic liveness and persistence properties. If we obtained these via pre-synthesis, then the idea does not always make sense: a single violation of a basic safety guarantee after some temporary assumption violation could allow the system to switch to some totally different state in the simulated Rabin(1) automaton, possibly allowing it to behave in a trivial manner for the rest of the computation.

Unfortunately, there is no approach to mitigate this problem in general, as the structure of the original specification is lost after pre-synthesis, and a meaningful ruggedisation of a specification in which a sub-property is broken down into multiple sub-sub-properties depends highly on the specification context. As an example, we could have the assumption in a specification that at precisely every second computation cycle, starting with cycle 0, some input bit p should be set to **true**. If during the execution of the system, the environment flips the phase of the signal (e.g., from setting the bit to **true** in every even cycle to setting it to **true** in every odd cycle), whether this should count as only a temporary violation or a permanent one depends on the application. Once the property is broken down into its sub-properties p and $\mathsf{G}(p \leftrightarrow \mathsf{X}\neg p)$, the phase shift can only count as a temporary assumption violation, even though depending on the application, it probably should not.

To obtain a reasonable ruggedisation of a specification, it is thus important to delay pre-synthesis until after ruggedisation. In Chapter 14.2, we combine the ideas of generalised Rabin(1) synthesis with the pre-synthesis-free ACTL $\cap$ LTL synthesis workflow to be presented in the next chapter. It will enable the meaningful application of robust synthesis to a larger set of safety properties in specifications. Of course, this approach cannot solve the problem that in the example above, it is unclear how a ruggedisation of the assumption that is suitable for some fixed but unknown practical application has to look like. Note, however, that in linear-time-temporal logic, which we use for expressing the specification, it is not possible to encode the requirement that precisely in every second computation cycle p should be set without connecting the values of p in two computation cycles, such as in $\mathsf{G}(p \leftrightarrow \mathsf{X}\neg p)$. This fact is rooted in the fact that LTL can only express *non-counting specifications*. Such an LTL expression then specifies that a phase shift shall only count as a temporary property violation, and thus already declares the designer's indent.

12.5 Improving the quality of the robust solution: error-free starts and bounded-transition-phase generalised Rabin(1) synthesis

We have discussed above how the introduction of stability properties to GR(1) synthesis makes the set of specifications that can be handled by the approach closed under ruggedisation. We can thus take the ruggedised specifications and hand them to an ordinary GRabin(1) synthesis procedure. There may however be more attributes that we want a robust implementation to have apart from convergence. One commonly required such attribute is that no guarantee violations should happen at all before a temporary assumption violation has been witnessed. Another possible attribute is that there must exist an upper time bound on the number of computation cycles between a temporary assumption violation and the last temporary guarantee violation afterwards (provided that no further temporary assumption violation occurs). This way, the environment cannot arbitrarily delay the return of the synthesised system to normal operation mode.

We discuss in this section how these two additional quality constraints can be taken into account in generalised Rabin(1) synthesis. In the first case, we solve the problem on the parity game level. We will see that the requirement that no temporary guarantee violation should occur prior to a temporary assumption violation comes for free when choosing a fixed-point based parity game solving algorithm. As these are well-suited for symbolic synthesis anyway, this choice of algorithm is natural. For the second case, we will see how we can adapt a specification to prevent arbitrarily long transition phases.

12.5.1 No temporary guarantee violation shall occur before a temporary assumption violation occurs

Let $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{init}, c)$ be a parity game. We define $\mathrm{Attr}_1 : 2^{V^0} \to 2^{V^0}$ to be the two-step enforceable predecessor operator for player 1. Given some set $X \subseteq V^0$, it computes from which set of player 0-positions player 1 can enforce that after two steps (i.e., after one move of player 0 and one move of player 1), the play can only be in a position in X. Using this operator, the winning positions in the game can be characterised by a nested fixed point equation (Emerson and Jutla, 1991; Dawar and Grädel, 2008). For the special case that we have here (i.e., only positions of player 0 can have a colour different to 0), that equation simplifies to:

$$\nu X_4.\mu X_3.\nu X_2.\mu X_1.\nu X_0. \bigcup_{i\in\{0,\dots,4\}} \left(c^{-1}(i) \cap \mathrm{Attr}_1(X_i)\right)$$

When looking at the formula, we see that there is a greatest fixed point operator for every even colour, and a least fixed point operator for every odd colour. The evaluation of the formula is easy to implement symbolically (e.g., with BDDs). After the computation, we can check if the initial position is contained in the result, and obtain the result that the underlying specification is unrealisable in case of a negative answer. Otherwise, we can obtain a winning strategy for the game by looking at the prefixed points of the equation. Note that as the greatest fixed point operators overapproximate the set of winning positions until a fixed point has been reached, we need to fully evaluate them before we have identified some positions that are surely winning. On the other hand, for the least fixed point operators, this is not the case. If we compute

$$W_{k,l} = \nu X_4.\mu^k X_3.\nu X_2.\mu^l X_1.\nu X_0. \bigcup_{i\in\{0,\dots,4\}} \left(c^{-1}(i) \cap \mathrm{Attr}_1(X_i)\right)$$

for some $k, l \in \mathbb{N}$, then $W_{k,l}$ can only contain winning positions. The lower the values of l and k in this context are, the earlier we have deduced that the positions contained in $W_{k,l}$ are winning, and thus the easier it is to win a game from these positions. This fact allows us to extract a strategy. When following the strategy, player 1 always plays the move that leads to a next position that appears as early in the list $W_{0,0}, W_{0,1}, \dots, W_{0,\infty}, W_{1,0}, W_{1,1}, W_{1,2}, \dots, W_{\infty,\infty}$ as possible. The correctness of this procedure follows from the original arguments by Emerson and Jutla (1991).

Recall that the only property types that can be violated temporarily are basic safety properties. After ruggedising the specification, in the GRabin(1) games, they are represented as stability properties. If there are no stability properties in the specification prior to ruggedisation, these are furthermore the only stability properties that we get then. In the initial position of a GRabin(1) game, the f_r flag is set to **true**. This ensures that a temporary guarantee violation will immediately lead to visiting a position with colour 4, and the first temporary assumption violation in a game leads to visiting a position with colour 3, unless in the same computation cycle, we also see a temporary guarantee violation, in which case colour 4 is visited instead.

For ensuring that no temporary guarantee violation can occur before a temporary assumption violation, we thus need a strategy that prevents visiting a position with colour 3 before a position with colour 4 is visited. It turns out that if we compute a strategy according to the method above, we obtain this property for free.

To see this, consider the following prefixed points:

$$W_{0,l} = \nu X_4.\mu^0 X_3.\nu X_2.\mu^l X_1.\nu X_0. \bigcup_{i\in\{0,\dots,4\}} \left(c^{-1}(i) \cap \mathrm{Attr}_1(X_i)\right)$$

The position set $W_{0,l}$ cannot contain any position with colour 3 by definition, but can contain positions with colours 0, 1, 2, and 4. Even more, since we evaluated the outer fixed point to the limit, X_4 describes the set of winning positions in the game. Thus, $\left(c^{-1}(4) \cap \mathrm{Attr}_1(X_4)\right)$ evaluates to all winning colour-4 positions. At the same time, positions with colour 3 are not part of X_2, X_1, and X_0 in the final valuation for these variables when computing $W_{0,\infty}$. Thus, $W_{0,\infty}$ describes the positions from which we can win the game when never visiting a position with colour 3, except for the case that we visit a position with colour 4 first. A corresponding strategy can be obtained by following the prefixed points $W_{0,0}, W_{0,1}, \dots, W_{0,\infty}$. As this sequence is a prefix of the sequence of pre-fixed points used to extract a strategy from a parity

game in the general case, there is actually nothing we need to do: if there exists a strategy that ensures that no temporary guarantee violation can happen before a temporary assumption violation, we will find it anyway. The only thing we have to do is to examine if the initial position is in the set $W_{0,\infty}$, because if it is not, we obtain a valid strategy for the specification, but there exists none that works without first temporarily violating some guarantee.

For the case that the specification from which a robust system is to be synthesised has some stability guarantees prior to ruggedisation, this approach is not immediately applicable, as we cannot see from visiting a position with colour 3 or 4 if the stability property $\mathsf{GF}\psi$ for which the position witnesses the non-satisfaction of ψ has been introduced in the ruggedisation process or not. Visiting a colour-3 position for a property that was already present in the non-ruggedised specification should be allowed before the first temporary safety guarantee violation. To solve this problem, we can extend the game by the two additional colours 5 and 6. We then use colour 5 for the stability guarantees that have been introduced in the ruggedisation process, and colour 6 for such stability assumptions. The f_r flag in the game also has to be duplicated then.

12.5.2 Bounded transition phases

A system synthesised from a ruggedised specification asserts that the transition phase to normal operation mode is bounded, but the length of this period might depend on the input. Consider the following example, which is a realisable specification over $\mathsf{AP}^I = \{i\}$ and $\mathsf{AP}^O = \{o\}$:

$$(i \wedge \mathsf{GF}\, i \wedge \mathsf{G}(i \leftrightarrow \mathsf{X}\, i)) \rightarrow (\mathsf{G}(o \leftrightarrow i) \wedge \mathsf{G}\, o)$$

The environment can temporarily violate its specification by switching from continuously choosing $i =$ **true** to continuously choosing $i =$ **false** and vice versa. While the environment plays a stream of $i =$ **false**, the system has to violate at least one of its guarantees. However, the environment has to switch back to setting $i =$ **true** at some point in order not to violate its liveness property. Thus, also the ruggedised version of the specification is realisable. However, we cannot give a time bound on the duration of a phase in which i is set to **false** continuously and thus, there is no implementation of the specification that has a time bound on the length of the *transition phase* in which the system switches back to normal operation mode after the last temporary assumption violation.

Unlike the case in Chapter 12.5.1, we do not obtain bounded transition phases for free with the standard fixed-point based parity game solving algorithm. We can however synthesise these by modifying the specification for the system to be synthesised a bit. Let us start by formally defining what the normal operation mode of a system is.

Definition 51. *Given a specification $\psi = (a_1 \wedge a_2 \wedge \ldots \wedge a_{n_a}) \rightarrow (g_1 \wedge g_2 \wedge \ldots \wedge g_{n_g})$, where all assumptions and guarantees are initialisation, basic safety, basic liveness or persistence properties, we say that a robust system is in normal operation mode with respect to ψ after the input/output prefix word $w \in (2^{\mathsf{AP}^I \uplus \mathsf{AP}^O})^*$ if the system can enforce that the postfix word $w' \in (2^{\mathsf{AP}^I \uplus \mathsf{AP}^O})^\omega$ representing the following input/output either has the property that the initialisation assumptions are not satisfied in ww', or no safety guarantee temporary violation is witnessed in ww' from position $|w| + 1$ onwards before the next safety assumption violation.*

We say that a system is in *recovery mode* whenever it is not in normal operation mode. Typically, systems that guarantee an upper time bound on the number of computation cycles being in recovery mode after a temporary assumption violation has occurred (provided that no further such violation occurs during the recovery process) are more desirable (Chatterjee et al., 2009) than those systems that do not provide such a guarantee. However, such systems do not always exist, as the example given above shows.

In this section, we show how to synthesise, whenever possible, systems that nevertheless give such a guarantee. As a side-result, the systems synthesised also have an additional output bit that always indicates whether normal operation mode has already been restored. As an example of a case in which such a bit is useful, consider a flight control system. In case of a temporary malfunction of another system connected to the flight control unit, the information that normal operation of the control unit has been restored is useful for the pilot when deciding whether to abort the flight or not.

To enforce that the transition phase is finite, we can ask the system to *announce* whether it is in that phase. We start with the non-ruggedised form of the specification for the system to be synthesised.

We extend the output propositions by an element t (for "transition phase"), and connect t with the assumptions and guarantees as follows:

- We add an initialisation guarantee $\neg t$.
- All liveness assumptions $\mathsf{GF}\psi$ are replaced by $\mathsf{GF}(\psi \vee t)$.
- All stability assumptions $\mathsf{FG}\psi$ are replaced by $\mathsf{FG}(\psi \vee t)$.
- All liveness guarantees $\mathsf{GF}\psi$ are replaced by $\mathsf{GF}(\psi \vee t)$.
- All stability guarantees $\mathsf{FG}\psi$ are replaced by $\mathsf{FG}(\psi \vee t)$.
- Let $\mathsf{G}\psi_1, \ldots, \mathsf{G}\psi_k$ be the basic safety assumptions. We replace them by a single safety guarantee $\mathsf{G}((\neg t \wedge \mathsf{X}t) \rightarrow \bigvee_{i\in\{1,\ldots,k\}} \neg\psi_i)$ and add the liveness property $\mathsf{GF}(\neg t \vee \bigvee_{i\in\{1,\ldots,k\}} \neg\psi_i)$ to the set of guarantees of the system. Thus, no basic safety assumption remains in the specification.
- We replace all basic safety guarantees $\mathsf{G}\psi$ by $\mathsf{G}(\psi \vee \mathsf{X}t)$.

Our modified specification now enforces bounded transition phases on the original specification. To see this, observe that t can only be set to **true** after witnessing a temporary safety assumption violation. Then, if these at some point stop occurring, the system has to return to normal operation mode after some time by the guarantee $\mathsf{GF}(\neg t \vee \bigvee_{i\in\{1,\ldots,k\}} \neg\psi_i)$ (unless safety assumption violations keep occurring). In the recovery phase, all liveness and stability properties and the safety guarantees are, in a sense, deactivated by the construction from above. A finite-state strategy thus needs to eventually declare the transition phase to be over. The number of computation steps until this happens without a safety assumption violation in between cannot exceed the number of positions in the game for positional strategies (which, w.l.o.g., we can assume the game solving algorithm that is employed in the synthesis process to compute) and is thus bounded. Only during the transition phase, safety guarantees may be violated temporarily.

Note that bounded reaction phases have been considered earlier by Chatterjee et al. (2009) for finitary parity games. Using the approach above, we however avoid that specifications in which the system to be synthesised is required to wait for some external events for fulfilling its obligations become unrealisable, which applies to the majority of the industrial case studies available for generalised reactivity(1) synthesis at the moment (see, e.g., Bloem et al., 2007a,b; Wongpiromsarn et al., 2010a). The encoding above is furthermore closely related to the sound but incomplete approach by Wongpiromsarn et al. (2010a) to incorporate stability properties as supported guarantee types into the generalised reactivity(1) synthesis approach, as it is based on the idea to have the system announce when stability has been reached. However, no connection to the assumptions to allow for robust synthesis is drawn in that work.

Chapter 13

ACTL ∩ LTL synthesis

The key problem that led to the development of reactive synthesis procedures that cannot deal with full linear time temporal logic (LTL) is *complexity*. Pnueli and Rosner (1989a) showed that the synthesis problem from LTL is 2EXPTIME-complete. By restricting the type of specifications that can be used, the complexity of GR(1) synthesis on the other hand is only exponential, as shown in Chapter 12.3. This reduction comes at a high prize: many specifications cannot be expressed directly, and pre-synthesis needs to be applied, which is highly non-trivial to do in a manner that does not degrade the performance of symbolically solving the resulting synthesis games.

These facts give rise to the question if there exists a bigger class of specifications that we could allow (without pre-synthesis) in a synthesis approach that still has an EXPTIME complexity and that at the same time is well-suited for performing synthesis symbolically. The answer to this question that we give in this chapter is positive, and the key element that we employ are *universal very-weak word automata* (*UVWs*). Our new approach can accommodate all specifications for which the assumptions and guarantees are representable in form of these automata. Universal very-weak automata intuitively make the obligations to the system (or the environment) explicit, as shown by the following example.

Figure 13.1 describes an automaton that represents the guarantees of a simple arbiter with the interface $\mathcal{I} = (\{r_1, r_2\}, \{g_1, g_2\})$, which we adapted from an example by Schewe and Finkbeiner (2007). Every run of the automaton starts in state q_0. If the arbiter gives two grants at the same time (i.e., we have $g_1 \wedge g_2$), we branch to state q_1, which is rejecting and that we can never leave again. Thus, the automaton rejects the execution of a system that gives two grants simultaneously at some point in the execution. If we obtain a request r_1 but the system does not give a grant g_1 at the same time, we transition to q_2, where we wait until a grant g_1 has been given. If the grant is never given, we stay in q_2, and the run is rejecting. The same idea is also implemented for the second grant.

If we consider the run tree that the automaton induces from a word, we can read off from the states of the automaton that are present in a level of the run tree what happened in the past. If state q_1 is present in a level, then this means that two grants have already been given at the same time on the word. Likewise, if q_2 is present in a level, then the last request r_1 has not been followed by some grant g_1 yet. In a way, the states thus encode *obligations* that the system still has to fulfil, where q_1 stands for the unfulfillable obligation.

Starting from universal very-weak automata for the assumptions and guarantees in a specification, we can easily build a synthesis game and solve it symbolically using binary decision diagrams (BDDs). The number of bits to allocate in BDDs is only linear in the number of states of the UVWs, and the fact that the

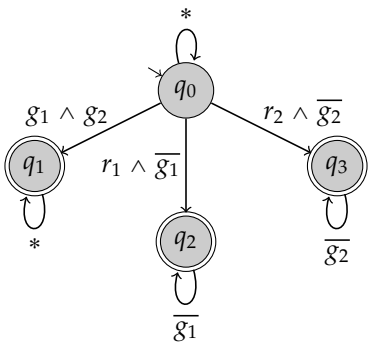

Figure 13.1: A universal very-weak automaton over infinite words that represents the guarantees to a simple arbiter with the interface $\mathcal{I} = (\{r_1, r_2\}, \{g_1, g_2\})$. Rejecting states are doubly-circled.

automata we consider are very-weak saves us from having to apply the Miyano-Hayashi construction (see Chapter 3.3) or a variant of it when building the game. Furthermore, as UVWs keep track of obligations for the system/environment, we can also replace BDDs as the symbolic reasoning engine with a different symbolic reasoning technique that has the potential of offering a better performance. Examples in this context are anti-chains (Filiot et al., 2009, 2010) or zero-suppressed binary decision diagrams (ZBDDs, Minato, 1993), a variant of BDDs that is well-suited for describing sets of clauses (Chatalic and Simon, 2001) that can in turn describe position sets in games.

Universal very-weak automata however have one problem: they are not so well suited as a specification formalism to be used by specification engineers. Thus, to allow their application in practical synthesis workflows, we must augment them with an automated translation algorithm from a specification formalism that is suitable for engineers. Maidl (2000) described UVWs to be the characterising automaton class for those properties of reactive systems that can be described both in *ACTL*, i.e., computation tree logic (CTL) with only universal path quantifiers, and LTL. This fact suggests to either use ACTL or LTL as higher-level specification language. Maidl gave a construction to translate an ACTL formula to a UVW, but just as UVWs, branching time logics are difficult to use for specifying system behaviour (Vardi, 2001).

So the alternative is to use LTL as specification formalism, but previously, it was unknown how to perform the translation from LTL to UVWs. We solve this problem in this chapter. As Maidl described how to translate from a UVW to an ACTL formula, we also obtain the first method to translate from LTL to ACTL, whenever possible.

Our translation algorithm accepts all LTL formulas that have equivalent UVWs, and we do not impose any restriction on the syntactic structure of the LTL formulas. Rather, the meaning of a formula is analysed, and we build the UVW from the information obtained in that step. Often, there are many ways to write down a specification, and it has been a point of criticism for reactive synthesis that the synthesised solutions typically depend on the way in which a specification is written (Madhusudan, 2011). Our translation algorithm operates by finding so-called *vermicelli* in the language of a deterministic automaton that we compute from the LTL formula. There is a fixed order on the vermicelli, which guarantees that we obtain the same UVWs for two LTL formulas that are equivalent, thus addressing the point of criticism by Madhusudan (2011).

We start this chapter by discussing the general properties of universal very-weak automata. Then, we describe how to translate an LTL formula to a very weak automaton, whenever possible, in Chapter 13.2. We apply the construction in Section 13.3, where we incorporate it into a synthesis workflow for "$\bigwedge$ Assumptions $\rightarrow$ $\bigwedge$ Guarantees" specifications, in which each assumption and guarantee is in ACTL ∩ LTL, i.e., the set of properties that are representable in both ACTL and LTL.

13.1 Very-weak automata

Most results on universal very-weak automata over infinite words (UVWs) are due to Maidl (2000). She identified them as being the characteristic automaton class for the set of tree properties that are representable in LTL and ACTL at the same time. This language class is called the intersection of LTL and ACTL, which is slightly imprecise due to the fact that ACTL properties are tree properties, but LTL properties are trace properties. By interpreting the LTL formula along all paths of a computation tree, and requiring the satisfaction along all of these, we however obtain a tree property, making the classes compatible.

Maidl described an algorithm to check for a given ACTL formula if it lies in the intersection. For the LTL case, Maidl defined a syntactic fragment of it, named LTL^{det}, whose expressivity coincides with that of ACTL ∩ LTL. However, she did not show how to translate an LTL formula into this fragment whenever possible, and the fragment itself is cumbersome to use, as it essentially requires the specification engineer to describe the structure of a UVW in LTL. Furthermore, there is not even a standard disjunction operation in LTL^{det}, although UVWs are closed under disjunction. Thus, for all practical means, the question how to check for a given LTL formula if it is contained in ACTL ∩ LTL remained open.

When synthesising a system, the designer of the system specifies the desired sequence of events, for which linear-time logics are more intuitive to use than branching-time logics. Thus, to use the advantage of universal very-weak automata in actual synthesis tool-chains, the ability to translate from LTL to a UVW is highly desirable.

Recently, Bojańczyk (2008) gave an algorithm for testing the membership of the set of models of an LTL formula in ACTL $\cap$ LTL after the LTL formula has been translated to a deterministic parity automaton. However, the algorithm cannot generate a universal very-weak automaton (UVW) from the parity automaton in case of a positive answer. The reason is that the algorithm is based on searching for so-called *bad patterns* in the automaton. If none of these are present, the deterministic parity automaton is found to be convertible, but we do not obtain any information about how a UVW for the property might look like.

Here, we solve this problem by reducing the problem of constructing a UVW for a given ω-regular language to a sequence of problems over automata for finite words. We modify a procedure by Hashiguchi (1983) that builds a distance automaton to check if a given language over finite words can be decomposed into a set of *vermicelli* (see Def. 53). Our modification adds a component to keep track of vermicelli already found. This way, by iteratively searching for vermicelli of increasing length in the language, we eventually find them all and obtain a full language decomposition.

13.1.1 Properties of very-weak automata

Let us now discuss properties of non-deterministic and universal very-weak automata, as the motivation and main idea of our approach is based on these. Given two automata, we call computing a third automaton that represents the set of words that are accepted by both automata *taking their conjunction*, while *taking their disjunction* refers to computing a third automaton that accepts all words that are accepted by either of the two input automata.

Definition 52. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a universal or non-deterministic automaton over finite or infinite words. We call $\mathcal{A}$ very-weak if there exists a partial order $\leqslant_Q$ over Q such that for all $q, q' \in Q$, if $q' \in \delta(q, x)$ for some $x \in \Sigma$, then $q \leqslant q'$. Furthermore, if $\mathcal{A}$ is an automaton over infinite words, then we require that $\mathcal{F}$ is a co-Büchi acceptance condition in case of universal branching, and a Büchi acceptance condition in case of non-deterministic branching.*

Recall that for a co-Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, we call $\mathcal{F}$ the set of rejecting states, whereas for a Büchi automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, we call $\mathcal{F}$ the set of accepting states. For automata over finite words $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$, we have that $\mathcal{F}$ is the set of accepting states, regardless of the branching condition.

Proposition 2. *Universal and non-deterministic very-weak automata over infinite and finite words are closed under disjunction and conjunction. Given two very-weak automata $\mathcal{A}$ and $\mathcal{A}'$ with state sets Q and Q', we can compute their disjunction and conjunction in polynomial time, with the following state counts of the results:*

1. *for universal automata and taking the conjunction: $|Q| + |Q'|$ states,*
2. *for non-deterministic automata and taking the disjunction: $|Q| + |Q'|$ states,*
3. *for universal automata and taking the disjunction: $|Q| \cdot |Q'|$ states, and*
4. *for non-deterministic automata and taking the conjunction: $|Q| \cdot |Q'|$ states.*

Proof. For the first two cases, the task can be accomplished by just merging the state sets, transitions, and sets of accepting/rejecting states. For cases 3 and 4, a standard product construction can be applied, with defining those states in the product as rejecting/accepting for which both corresponding states in the factor automata are rejecting/accepting, respectively (Janin and Lenzi, 2004). □

Proposition 3. *Universal or non-deterministic very-weak automata over finite or infinite words are not closed under complementation.*

Proof. For the finite-word case, consider the language $L = \Sigma^* \backslash \Sigma^* aa \Sigma^*$. It can be represented as a universal very-weak automaton, which we depict in Figure 13.2, but not as a non-deterministic one: we cannot have a self-loop on a at any state that is reachable and from which an accepting state (with some self-loop) can be reached, as otherwise we can construct an accepting run that takes this loop on a twice in succession and then moves to the accepting state with the self-loop and follows it infinitely often, and thus the automaton would accept too many words. But then, at any reading of a, we must proceed to some next state in the automaton along an accepting run. Thus, after reading (ab) as often as there are

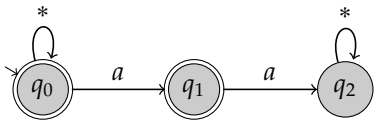

Figure 13.2: A universal very-weak automaton over finite words for the language $L = \Sigma^*\backslash\Sigma^* aa\Sigma^*$

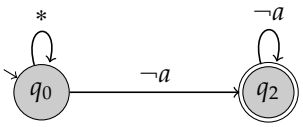

Figure 13.3: A universal very-weak automaton over infinite words for the LTL property $\mathsf{GF}\, a$

states in the automaton, all runs must have ended, and we cannot accept a word starting this way. Thus, the automaton can only accept too few or too many words. As by duality, universal very-weak and non-deterministic very-weak automata are capable of representing the respective complement languages, the claim follows for automata over finite words.

For the infinite-word case, there are several ways to prove the claim. A simple proof is to use the fact that the LTL property $\mathsf{GF}a$ is representable as a universal very-weak automaton (as depicted in Figure 13.3), but it is well-known that the LTL property $\mathsf{FG}\neg a$, which is the complement of $\mathsf{GF}a$, is not representable in ACTL, and thus by the results of Maidl (2000), it is also not representable as a universal very-weak automaton. By duality, the same argument also serves as proof for the case of non-deterministic very-weak automata over infinite words. □

Since very-weak automata are closed under conjunction and disjunction, the question if alternating very-weak automata are equivalent to universal or non-deterministic very-weak automata is natural. For both the cases of infinite and finite words, this conjecture does not hold.

Alternating very-weak automata are easily complementable (by replacing universal branching with non-deterministic branching and complementing the set of rejecting states), so this conjecture contradicts Proposition 3. This fact might be surprising at first, as closure under disjunction and conjunction would seem to imply that alternating very-weak automata have the same expressive power. Intuitively, this assumption does however not hold as on a self-loop in a very-weak automaton, we take the disjunction (in case of non-deterministic branching) or conjunction (in case of universal branching) of an infinite set of languages, and the automata are only closed under taking finitely many disjunctions or conjunctions.

Proposition 4. *Every very-weak automaton has an equivalent one of the same type for which no accepting/rejecting state has a non-self-loop outgoing edge (called the* separated form *of the automaton henceforth).*

Proof. Duplicate every accepting/rejecting state in the automaton and let the duplicate have the same incoming edges. Then, mark the original copy of the state as non-accepting/non-rejecting. The left part of Figure 13.4 shows an example of such a state duplication. □

The fact that every automaton has a separated form allows us to decompose it into a set of so-called *simple chains*:

Definition 53. *Given an alphabet Σ, we call a subset Q' of states of an automaton over Σ a* simple chain *if there exists a transition order on Q', i.e., an bijective function $f : Q' \to \{1, \ldots, |Q'|\}$ such that:*

- *only the state q with $f(q) = 1$ is initial,*
- *only the state q with $f(q) = |Q'|$ is accepting/rejecting,*
- *there is no transition in the automaton between a state in Q' and a state that is not in Q',*
- *for every transition from q to q' in the automaton, $f(q) \leqslant f(q') \leqslant f(q) + 1$.*

Furthermore, regular expressions that are unnested concatenations of elements of the form A, A^*, and A^ω for $A \subseteq \Sigma$ are called vermicelli.

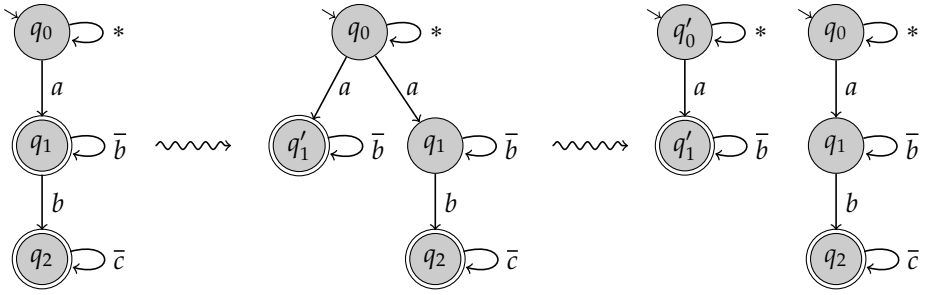

Figure 13.4: Example for converting an UVW into separated form and subsequently decomposing it into simple chains. The automata in this example are equivalent to the LTL formula $\mathsf{G}(a \rightarrow \mathsf{XF}(b \wedge \mathsf{XF}c))$. We use Boolean combinations of atomic propositions and their negation as edge labels here. For example, $\overline{b}$ refers to all elements $x \in \Sigma = 2^{\mathsf{AP}}$ for which $b \notin x$. Rejecting states are doubly-circled.

As an example, the right-most sequence of states in Figure 13.4 is a simple chain and can equivalently be represented as the vermicelli $\Sigma^* a(\overline{b})^* b(\overline{c})^\omega$. Note that every vermicelli can be translated to a language-equivalent set of simple chains.

Proposition 5. *Every very-weak automaton can be translated to a form in which it only consists of simple chains.*

Proof. Convert the very-weak automaton into separated form and enumerate all paths to leaf states along with the self-loops that might possibly be taken. For every of these paths, construct a simple chain. □

Let us conclude by establishing the relationship of the expressiveness levels of UVWs and deterministic Büchi automata.

Proposition 6. *Every UVW is representable as a deterministic Büchi automaton.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a UVW. We extend $\delta : Q \times \Sigma \to 2^Q$ to a function $\hat{\delta} : 2^Q \times \Sigma \to 2^Q$ such that for all $S \subseteq Q$ and $x \in \Sigma$, we have $\hat{\delta}(S, x) = \{q' \in Q \mid \exists q \in S : q' \in \delta(q, x)\}$. Without loss of generality, we assume that $\mathcal{F} = \{q_1, \ldots, q_m\}$ for some $m \in \mathbb{N}$. We can construct an equivalent deterministic Büchi automaton $\mathcal{A}' = (Q', \Sigma, \delta', Q'_0, \mathcal{F}')$ as follows:

$$
\begin{aligned}
Q' &= 2^Q \times \{0, \ldots, m\} \\
\forall (S, 0) \in Q', x \in \Sigma : \delta'((S, 0), x) &= (\hat{\delta}(S, x), 1) \\
\forall (S, j) \in Q', x \in \Sigma \text{ with } j \geqslant 1 \text{ and } q_j \notin \delta(q_j, x) \vee q_j \notin S : \delta'((S, j), x) &= (\delta(S, x), (j + 1) \bmod m) \\
\forall (S, j) \in Q', x \in \Sigma \text{ with } j \geqslant 1 \text{ and } q_j \in \delta(q_j, x) \wedge q_j \in S : \delta'((S, j), x) &= (\delta(S, x), j) \\
Q'_0 &= \{(Q_0, 0)\} \\
\mathcal{F}' &= \{(S, 0) \mid S \subseteq Q\}
\end{aligned}
$$

Assume that for a given word $w \in \Sigma^\omega$, there exists some rejecting state q in $\mathcal{A}$ that is eventually entered along some run and never left from that point onwards, where we say that some state q is left along a run at position i if $q \notin \delta(q, w_i)$ for the word $w = w_0 w_1 \ldots$ to which the run corresponds, or if the run is not in q at position i. The construction ensures that then, the counter value cannot be increased from j to $j + 1$ (or 0 if $j = m$) in the corresponding run of $\mathcal{A}'$ any more. As a consequence, $\mathcal{A}'$ also rejects the word as the counter value 0 cannot occur along the run from that time onwards.

On the other hand, if a word is accepted by the UVW, then from every position in every corresponding run, every rejecting state in the UVW is eventually left. Since the counter values can increase precisely whenever this happens, we get a value of 0 infinitely often, leading to visiting accepting states in the deterministic Büchi automaton infinitely often. □

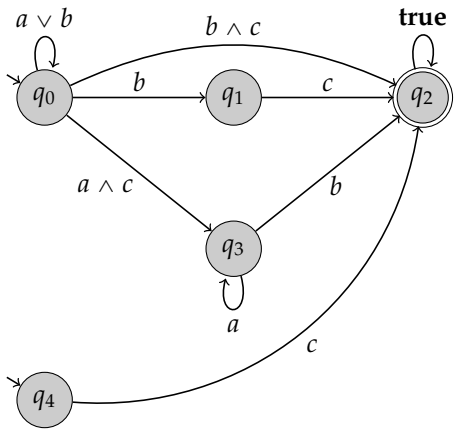

Figure 13.5: A non-deterministic very-weak automaton over infinite words for the LTL property $(a \mathsf{U} b) \mathsf{U} c$

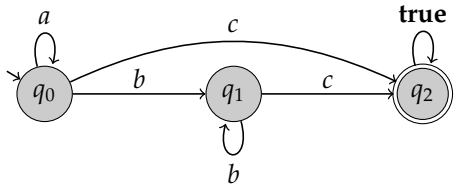

Figure 13.6: A non-deterministic very-weak automaton over infinite words for the LTL property $a \mathsf{U} (b \mathsf{U} c)$

13.2 Obtaining universal very-weak automata from languages

To make a synthesis approach that harnesses the simplicity of universal very-weak automata over infinite words work, we must of course have a way to obtain them. Typically, we start from linear-time temporal logic (LTL) when trying to obtain such automata.

It turns out that current techniques to translate from LTL to non-deterministic Büchi automata do not produce very-weak solutions whenever possible – if this were true, we could, by duality, use such a technique to obtain a UVW by complementing the LTL formula.

A good example in this context are until-chains: for the LTL formula $(a \mathsf{U} b) \mathsf{U} c$, the LTL-to-Büchi translator `ltl2ba 1.1` (Gastin and Oddoux, 2001) does not compute a non-deterministic very-weak automaton, which could then be used as a UVW for $\neg((a \mathsf{U} b) \mathsf{U} c)$, although there exists one (depicted in Figure 13.5).

For other formulas, such as $a \mathsf{U} (b \mathsf{U} c)$, a very-weak automaton is however found by `ltl2ba` (shown in Figure 13.6). This example demonstrates that specialised techniques geared towards UVWs are needed, but we may use a standard LTL-to-Büchi converter as an incomplete method when checking if the result happens to be very-weak.

Since alternating very-weak automata are the characterising automaton class for LTL, a first candidate for a specialised technique is to apply a translation technique that recurses over sets of states in the alternating automaton and performs alternation removal. Such an approach is however infeasible from a semantic point of view: if we conclude for a sub-branch in the alternating automaton that there exists no equivalent non-deterministic or universal very-weak automaton, then this does not mean that for the overall alternating automaton, this fact also holds. Take for example the LTL formula $\mathsf{GF}a \vee \mathsf{FG}\neg a$. It is equivalent to **true**. However, one of its disjunctions is not representable as non-deterministic very-weak automaton, and the other one is not representable as universal very-weak automaton. Thus, when trying to non-determinise or universalise the alternating automaton for this formula, the process fails. Classical approaches to alternation removal (e.g., for obtaining a non-deterministic Büchi automaton) do not have this problem, as they are applied to cases for which it is guaranteed that a suitable result automaton exists. For fixing this problem, we would need to track in which states in other branches of

a run tree we could be at the same time. But then, as using such a power set construction can introduce cycles in the reachability graph, it is hard to obtain a very-weak automaton that tracks such information. As a summary, it is fair to say that conceptually, this line of thought does not appear to be promising.

As automata over finite words are a classical topic in the computer science literature and even better studied, a closer look at results on these to see if they help us with the present problem is worthwhile, as we can find a way to decompose the problem of obtaining a UVW for a language over infinite words to a set of problems over finite words. For the finite-word case, we are then either interested in finding a non-deterministic very-weak automaton or a universal very-weak automaton, both over finite words. If we start from a deterministic automaton over finite words (DFA), the problems can be considered to be equivalent, as by duality, we can simply complement the input language (without blowing up the DFA) whenever we are interested in the respective other model. Non-deterministic branching is however the predominant branching mode in the literature, so let us take that viewpoint now.

The *star height* of a language over finite words describes the maximum number of nestings of the star-operator needed to represent the language as a regular expression. When examining non-deterministic very-weak automata over finite words, it seems apparent that the star height of their languages might be at most one. However, it is currently unknown if computing the star height of a language is actually decidable, and it is equally unknown how to compute a regular expression for such a language in which no star-operator occurs in the operand of another star-operator.

Another closely related language class are the so-called *star-free languages*, which are generated by letters in the alphabet and the concatenation, union, and complementation operations. These are however more expressive than the languages representable by universal or non-deterministic very-weak automata and thus not of relevance here. An example is the language $\Sigma^* \setminus \Sigma^* aa \Sigma^*$, which is not representable as a non-deterministic very-weak automaton (see the proof of Proposition 3), but which can be written as the star-free expression $\overline{\overline{\epsilon} aa \overline{\epsilon}}$. Obviously, star-free languages are also closed under complementation, which the language class of interest here is not.

Hashiguchi (1983) considered classes of languages that can be built from a set of ground languages and applying a restricted subset of the concatenation, union, and star operators. If we take as set of ground languages $\{X, X^* \mid X \subseteq \Sigma\}$ for some alphabet Σ, and allow the concatenation and union operators, we capture precisely the set of languages representable as non-deterministic very-weak automata over the words in Σ^*. Hashiguchi (1983) gave a procedure for checking if a language, given as deterministic automaton over finite words, can be represented in this way, and preceded a construction for the same purpose (but restricted to the class of languages that are representable by non-deterministic very-weak automata over finite words) by Bojańczyk (2008) by about 25 years. Both procedures are non-constructive, i.e., they only check if a language is representable in this way, but do not provide such a representation in case of a positive answer. Bojańczyk (2008) uses a search for bad patterns in the automaton, which leads to polynomial complexity of his algorithm. Hashiguchi (1983) on the other hand builds a *distance automaton* from the problem, which accepts every accepted word with a finite distance if and only if the language is representable as a combination of the ground languages using only the operators allowed. The complexity of this procedure is however much higher. Also, checking the finite distance of all words in the distance automaton is done by reducing the problem to determining if a set of basis vectors with elements in some *tropical algebra* spans a finite or infinite space, and is thus highly non-trivial. However, the procedure by Hashiguchi (1983) has one advantage: it **can** be made constructive, and we will see below how this is done. In fact, we will not even need any arguments from tropical algebra.

Using the procedure to obtain a non-deterministic very-weak automaton over finite words from some DFA, we can then construct an approach to obtain a UVW from some other automaton representation, say, a deterministic Büchi automaton. Recall that these can represent all properties describable as UVWs, and are thus sufficient in this context. The key insight needed for this task is that we can enumerate the possible rejecting states with their end-loops in a very-weak automaton over infinite words in separated form and for every of these states, compute a non-deterministic very-weak automaton over finite words for the words that should lead to this state. Let us have a look at this approach in more detail now.

13.2.1 The case of automata over infinite words

We have seen that every UVW can be translated to a separated UVW. In a separated UVW, we can distinguish rejecting states by the set of alphabet symbols for which the states have self-loops. If two

rejecting states have the same set, we can merge them without changing the language of the automaton. As a corollary, we obtain that a UVW can always be modified such that it is in separated form and has at most $2^{|\Sigma|}$ rejecting states. We will see in this section that obtaining a UVW for a given language L over some alphabet Σ can be done by finding a suitable *decomposition* of the set of words that are not in L among these up to $2^{|\Sigma|}$ rejecting states, and then constructing the rest of the UVW such that words that are mapped to some rejecting state in the decomposition induce runs that eventually enter that rejecting state and stay there forever.

Definition 54. *Given a language L over infinite words from the alphabet Σ, we call a function $f : 2^\Sigma \to 2^{\Sigma^*}$ an end-component decomposition of L if $L = \Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma}(f(X) \cdot X^\omega)$. We call f a maximal end-component decomposition of L if for every $X \subseteq \Sigma$, we have $f(X) = \{w \in \Sigma^* \mid w \cdot X^\omega \cap L = \varnothing\}$.*

Definition 55. *Given a separated UVW $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$ and an end-component decomposition f, we say that f corresponds to $\mathcal{A}$ if for $(q_1, X_1), \ldots, (q_m, X_m)$ being the rejecting states and alphabet symbols under which they have self-loops, we have:*

- *for all $i \neq j$, $X_i \neq X_j$;*
- *for all $X \subseteq \Sigma$ with $X \notin \{X_1, \ldots, X_m\}$, we have $f(X) = \varnothing$.*
- *Let $\hat{\delta} : 2^Q \times \Sigma \to 2^Q$ be the extension of δ such that for all $S \subseteq Q$ and $x \in \Sigma$, we set $\hat{\delta}(S, x) = \{q' \in Q \mid \exists q \in S : q' \in \delta(q, x)\}$. For $1 \leqslant i \leqslant m$, we have $f(X_i) = \{w_0 w_1 \ldots w_k \in \Sigma^* \mid q_i \in \hat{\delta}(\ldots \hat{\delta}(\hat{\delta}(Q_0, w_0), \ldots), w_k)\}$.*

As an example, the end-component decomposition that corresponds to the UVW in the middle part of Figure 13.4 is a function f with $f(\overline{b}) = \Sigma^* a(\overline{b})^*$, $f(\overline{c}) = \Sigma^* a(\overline{b})^* b(\overline{c})^*$, and $f(X) = \varnothing$ for $X \neq \overline{b}$ and $X \neq \overline{c}$. The decomposition is not maximal as, for example, the word $\{a\}\varnothing^\omega$ is not in the language of the automaton, but we have $\{a\} \notin f(\{\varnothing\}) = \varnothing$.

By the definition of corresponding end-components, every separated UVW for which we have merged rejecting states with the same self-loops has one unique corresponding end-component decomposition. Likewise, every language has one maximal end-component decomposition. The key result that allows us to reduce finding a UVW for a given language to a problem on finite words combines these two facts:

Lemma 16. *Let L be a language that is representable by a universal very-weak automaton. Then, L is also representable as a separated UVW whose corresponding end-component decomposition is the maximal end-component decomposition of L.*

Proof. Let a UVW be given whose end-component decomposition f is not maximal. The decomposition can be made maximal by taking $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$, without changing the language. To support this claim, we need to prove that (1) f' is an end component decomposition for the same language, and (2) that f' is maximal, which we do below. Then, building a corresponding UVW only requires taking disjunctions of parts of the original UVW. Since we know that UVW are closed under disjunction, it is assured that there also exists a UVW that corresponds to f'. It remains to prove the two sub-claims mentioned earlier in this paragraph.

1. Consider an end-component decomposition $\hat{f}$ that we obtain from f by setting $\hat{f}(X_2) = f(X_1) \cup f(X_2)$ for some $X_1 \subseteq \Sigma$ and $X_2 \subseteq \Sigma$ with $X_2 \subseteq X_1$, and $\hat{f}(X') = f(X')$ for all $X' \neq X_2$. Thus, $\hat{f}$ is obtained from f by merging $f(X_1)$ into $f(X_2)$. As f' with $f'(X) = \bigcup_{X' \supseteq X} f(X')$ for every $X \subseteq \Sigma$ can be obtained from f by applying finitely many such merging operations, if we can prove that f and $\hat{f}$ are end-component decompositions of the same language, then we also know this for f and f'.

   As $X_2 \subseteq X_1$, we have that:

   $$\hat{f}(X_2) \cdot X_2^\omega = (f(X_1) \cup f(X_2)) \cdot X_2^\omega \subseteq f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$$

   By taking the union with $f(X_1) \cdot X_1^\omega$ in every part of this equation (and removing the middle part), we obtain:

   $$f(X_1) \cdot X_1^\omega \cup \hat{f}(X_2) \cdot X_2^\omega \subseteq f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$$

   As by the definition of $\hat{f}(X_2)$, we have that $\hat{f}(X_2) \cdot X_2^\omega \supseteq f(X_2) \cdot X_2^\omega$, it follows that $f(X_1) \cdot X_1^\omega \cup \hat{f}(X_2) \cdot X_2^\omega = f(X_1) \cdot X_1^\omega \cup f(X_2) \cdot X_2^\omega$. We can now deduce that f and $\hat{f}$ are end-component decompositions of the same language:

   $$\Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma} (f(X) \cdot X^\omega)$$

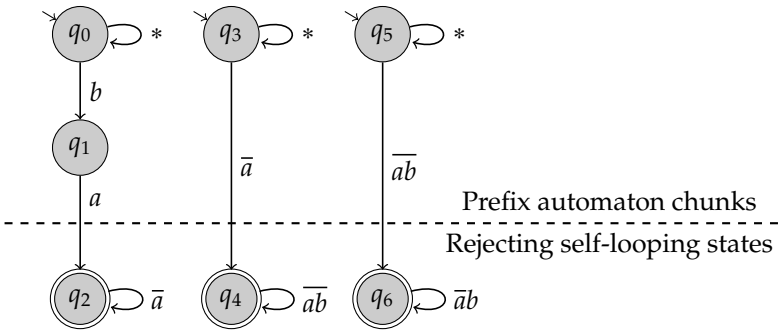

Figure 13.7: Example UVW in separated form for the LTL property $\mathsf{G}(b \wedge \mathsf{X}a \rightarrow \mathsf{XXF}a) \wedge \mathsf{G}(\neg a \rightarrow \mathsf{XF}(a \vee b)) \wedge \mathsf{G}(\neg a \wedge \neg b \rightarrow \mathsf{XF}(a \vee \neg b))$

$$
\begin{aligned}
&= \Sigma^\omega \setminus \left(\left(\bigcup_{X \subseteq \Sigma, X \neq X_1, X \neq X_2} (f(X) \cdot X^\omega) \right) \cup (f(X_1) \cdot X_1^\omega) \cup (f(X_2) \cdot X_2^\omega) \right) \\
&= \Sigma^\omega \setminus \left(\left(\bigcup_{X \subseteq \Sigma, X \neq X_1, X \neq X_2} (f(X) \cdot X^\omega) \right) \cup (f(X_1) \cdot X_1^\omega) \cup (\hat{f}(X_2) \cdot X_2^\omega) \right) \\
&= \Sigma^\omega \setminus \bigcup_{X \subseteq \Sigma} (\hat{f}(X) \cdot X^\omega)
\end{aligned}
$$

2. Let us now prove that f' is a maximal decomposition. Let $w \in \Sigma^*$ and $T = \{X_1, \ldots, X_n\}$ be the subsets of Σ such that for all $1 \leqslant i \leqslant n$, $w \cdot X_i^\omega \cap L = \varnothing$, for L being the language such that f' is an end-component decomposition of it. Observe the following facts:
   - First of all, T is a downwards-closed set, i.e., for all $X \in T$ and $X' \subseteq X$, we have $X' \in T$. This follows directly from the fact that $w \cdot X^\omega \supseteq w \cdot X'^\omega$ for all $X' \subseteq X$.
   - Furthermore, for all maximal elements $Y \in T$ (i.e. there exists no $X \supseteq Y$ with $X \in T$), we have that $w \in f(Y)$. This follows from the fact that for $Y = \{x_1, \ldots, x_m\}$, we have $w(x_1 x_2 \ldots x_m)^\omega \notin L$ by definition, and thus for some $X \supseteq Y$, we must have $w \in f(X)$ in order for the end component decomposition not to miss that $w(x_1 x_2 \ldots x_m)^\omega \notin L$. However, as for all $X \supset Y$, by the definition of T, we also have $w \cdot X_i^\omega \cap L \neq \varnothing$, having $w \in f(X)$ would lead to $L \neq \Sigma^\omega \setminus \bigcup_{X' \subseteq \Sigma} (f(X') \cdot X'^\omega)$. Thus, the only way for f to be a valid end-component decomposition for L is to have $w \in f(Y)$.

   As we have established that T is a downward-closed set and for all maximal elements Y, we have $w \in f(Y)$, we know that for all $X \subseteq \Sigma$, we have $X \in T$ (and thus $w \cdot X_i^\omega \cap L = \varnothing$) if and only if $w \in \bigcup_{X' \supseteq X} f(X')$. Since this line of reasoning holds for all $w \in \Sigma^*$, it follows that $\{w \in \Sigma^* \mid w \cdot X^\omega \cap L = \varnothing\} = \bigcup_{X' \supseteq X} f(X')$, and thus, f' is a maximal end-component decomposition.

□

Thus, in order to obtain a UVW for a given language $L \subset \Sigma^\omega$, we can compute the maximal end-component decomposition f' of L, and for every end component $X \subseteq \Sigma$, compute a non-deterministic very-weak automaton over finite words for $f'(X)$.

Let us visualise this fact for clarity. Assume that we have an alphabet $\Sigma = \{\overline{ab}, a\overline{b}, \overline{a}b, ab\}$, and we want to obtain a UVW for the LTL property $\mathsf{G}(b \wedge \mathsf{X}a \rightarrow \mathsf{XXF}a) \wedge \mathsf{G}(\neg a \rightarrow \mathsf{XF}(a \vee b)) \wedge \mathsf{G}(\neg a \wedge \neg b \rightarrow \mathsf{XF}(a \vee \neg b))$. We know that there exists a UVW for that property if and only if there exists such a UVW for it that is in separated form and for which all rejecting states with the same self-loop labels have been merged. All self-loop labels are subsets of Σ. We can ignore the self-loop label set $\varnothing$, as no word can be rejected with it. Thus, we can have at most $2^{|\Sigma|} - 1$ rejecting states. What we now have to do is to compute non-deterministic very-weak word automata that lead to these rejecting states such that when connecting these automata to the rejecting end-states, we obtain a UVW that rejects all words that do not satisfy the property along some of its rejecting states, but accepts all other words. Figure 13.7 shows an example of such an automaton. We can see that in a UVW in separated form, the words that are rejected by the automaton are *distributed* among the rejecting self-looping states. For example, the word $\{b\}\{a\}\{b\}^\omega$ is

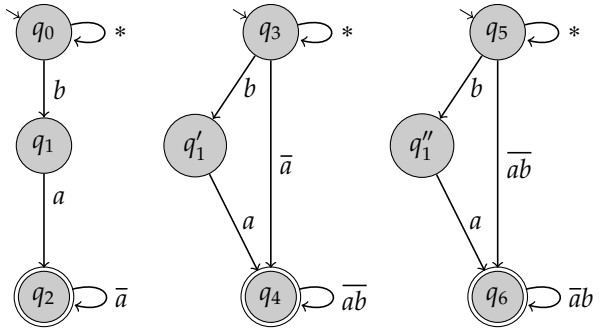

Figure 13.8: A UVW with the same language as the one in Figure 13.7, but with maximal end-component decomposition.

rejected by the left-most part of the automaton, whereas the word $\varnothing^\omega$ is rejected by the middle part of the automaton. For obtaining a UVW for an LTL property, we must find such a distribution of the words that do not satisfy the property among the rejecting self-looping states, which we called the end-component decomposition above. Finding a distribution that can be realised in a UVW is however not trivial. For example, for the language $\mathsf{GF}a \wedge \mathsf{GF}b$, we need two end components, and allocating all words that end with $\{\overline{ab}, \overline{a}b\}^\omega$ to the end component with self-loop $\overline{a}$, and allocating only the remaining words to be rejected to the end component with self-loop $\overline{b}$ cannot work - there is no way to prevent the automaton part for the second end component from rejecting some word that ends with $\varnothing^\omega$ while not rejecting too few words. Thus, in our distribution of words that do not satisfy the property along the end components, we must be more inclusive, and the maximal decomposition takes this idea to the maximum. We can view the prefix automaton chunks as NVWFs (non-deterministic very-weak automata over finite words) by replacing the rejecting states by an accepting state without outgoing edges. If we have a UVW that does not have a maximal end-component decomposition, then we can make it maximal by adding the words of one end component $X \subseteq \Sigma$ to another end component $X' \subseteq X$ by just taking the disjunction of the two NVWFs for the incoming words for X and X' and replacing the NVWF for X' by the result. By iterating this process until there is nothing more to add, we obtain a UVW with a maximal end-component decomposition. The result is shown in Figure 13.8.

Starting with an LTL formula, we can thus translate it to a UVW (if possible) as follows: first of all, we translate the LTL formula to a deterministic Büchi automata (see, e.g., Ehlers, 2010b for an overview). Note that as the expressivity levels of LTL and deterministic Büchi automata are incomparable, this is not always possible. If no translation exists, we however know that there also exists no UVW for the LTL formula, as all languages representable by UVWs are also representable by deterministic Büchi automata. After we have obtained the Büchi automaton, we compute for every possible end-component $X \subseteq \Sigma$ from which states S in the automaton every word ending with X^ω is rejected. This is essentially a model checking problem over an automaton with Büchi acceptance condition. By interpreting the deterministic Büchi automaton as a DFA, and making precisely those states in the DFA accepting from which every word in X^ω is rejected in the Büchi automaton, we obtain a deterministic automaton over finite words with S as the set of accepting states. This automaton is then used as input to a method to find a NVWF for the prefix language, i.e., the words under which the end-component $X \subseteq \Sigma$ shall be reachable from an initial state in the UVW to be computed.

Note that if the prefix language is the same for two different $X, X' \subseteq \Sigma$ with $X' \subset X$, we do not need to consider X' as an end component as the part of the UVW that we compute for X is guaranteed to already cover all words that are to be rejected at end component X'. Also, if the deterministic Büchi automaton represents a safety language (which is the case whenever the language of the automaton is empty or there exists no reachable loop without an accepting state), then we only need to consider $X = \Sigma$.

13.2.2 The case of automata over finite words

As a sub-procedure to be called by the construction to obtain UVWs from deterministic Büchi automata, we now establish a method of obtain NVWFs from deterministic automata over finite words. As

we discussed earlier, we modify a construction by Hashiguchi (1983) here to perform this task in a constructive manner.

Intuitively, we build a *distance automaton* over finite words that accepts words in the language of the input DFA, and from an accepting run for a word, we can read off a vermicelli that contains the word and such that all words in the language of the vermicelli are in the language of the DFA. A distance automaton behaves like a DFA, but every transition also has a cost (i.e., we have $\delta : Q \times \Sigma \to Q \times \mathbb{N}$, and the cost of a run is the sum of the costs of the individual transitions taken. Here, the cost of a run corresponds precisely to the length of the vermicelli witnessed by the run, and every transition has a cost in $\{0, 1\}$. We always search in the automaton for a vermicelli of minimal length, and thus avoid unnecessary blowup of the automaton. Whenever we find a vermicelli, we update a candidate NVFW for the simple chains found so far with the vermicelli, and then also update the distance automaton to exclude all vermicelli that we have found so far.

Definition 56. *Given a DFA $\mathcal{A} = (Q^{\mathcal{A}}, \Sigma, Q_0^{\mathcal{A}}, \delta^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}})$ for the language to be analysed and a NVWF $\mathcal{B} = (Q^{\mathcal{B}}, \Sigma, Q_0^{\mathcal{B}}, \delta^{\mathcal{B}}, \mathcal{F}^{\mathcal{B}})$ for the vermicelli already found, the non-deterministic vermicelli-searching distance automaton over finite words $\mathcal{D} = (Q, \Sigma, Q_0, \delta, F)$ is defined as follows:*

$$
\begin{aligned}
Q &= 2^{Q^{\mathcal{A}}} \times 2^{\Sigma} \times \mathbb{B} \times 2^{Q^{\mathcal{B}}} \\
Q_0 &= \{(Q_0^{\mathcal{A}}, \varnothing, \textbf{false}, Q_0^{\mathcal{B}})\} \\
\mathcal{F} &= \{(S, X, b, R) \mid S \subseteq \mathcal{F}^{\mathcal{A}}, (R \cap \mathcal{F}^{\mathcal{B}}) = \varnothing\} \\
\delta((S, X, b, R), x) &= \{((S, X, b, R'), 0) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X, b = \textbf{true}\} \\
&\cup \{((S', X', \textbf{true}, R'), 1) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X', S' = \hat{\delta}^{\mathcal{A},*}(S, X')\} \\
&\cup \{((S', X', \textbf{false}, R'), 1) \mid R' = \hat{\delta}^{\mathcal{B}}(R, \{x\}), x \in X', S' = \hat{\delta}^{\mathcal{A}}(S, X')\} \\
&\textit{for all } (S, X, b, R) \in Q, x \in \Sigma
\end{aligned}
$$

In this definition, we used the concept of extending some transition function δ of an automaton to a function $\hat{\delta}$ that handles transitions from multiple predecessor states. In contrast to the previous occurrences of this concept in this chapter, we also support multiple alphabet characters here. So for the transition function of some automaton δ, we define $\hat{\delta} : 2^Q \times 2^\Sigma \to 2^Q$ such that for all $S \subseteq Q$ and $X \subseteq \Sigma$, we have $\hat{\delta}(S, X) = \{q' \in Q \mid \exists q \in S, x \in X : q' \in \delta(q, x)\}$. Furthermore, $\hat{\delta}^* : 2^Q \times 2^\Sigma \to 2^Q$ denotes the multi-step version of δ, i.e., we define $\hat{\delta}^*(S, X) = S \cup \hat{\delta}(S, X) \cup \hat{\delta}(\hat{\delta}(S, X), X) \cup \hat{\delta}(\hat{\delta}(\hat{\delta}(S, X), X), X) \cup \ldots$ for all $S \subseteq Q$ and $X \subseteq \Sigma$.

The states in a vermicelli-searching automaton $\mathcal{D}$ are four-tuples (S, X, b, R) such that X and b represent an element in a vermicelli, where b tells us if the current vermicelli element X is starred. During a run, we track in S in which states in $\mathcal{A}$ we can be in after reading some word that is in the language of a vermicelli that consists of the vermicelli elements observed in the X and b state components along the run of $\mathcal{D}$ so far. Whenever we have $S \subseteq \mathcal{F}^{\mathcal{A}}$, then we know that all these words are accepted by $\mathcal{A}$. At the same time, the R component simulates all runs of the NVWF $\mathcal{B}$, and the definition of $\mathcal{F}$ ensures that no word that is in the language of $\mathcal{B}$ is accepted by $\mathcal{D}$. Thus, $\mathcal{D}$ can only find vermicelli that contain some word that is not accepted by $\mathcal{B}$. Transitions with cost 1 represent moving on to the next vermicelli element.

The following theorem captures the important properties of vermicelli-searching distance automata.

Theorem 14. *Let $\mathcal{A}$ be an DFA, $\mathcal{B}$ be a NVWF and $\mathcal{D}$ be the corresponding vermicelli-searching distance automaton. We have:*

- $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{A}) \backslash \mathcal{L}(\mathcal{B})$
- *Let $\mathcal{L}(\mathcal{A})$ contain a vermicelli $V = A_1 \ldots A_k$, where every A_i is either of the form X^* or X for some $X \subseteq \Sigma$. If V is not covered by $\mathcal{L}(\mathcal{B})$, then $\mathcal{D}$ accepts some word w that is a model of V with a run of distance k. Along this run, the first three state components only change during transitions with a cost of 1, and the second and third components in between changes describe the alphabet symbol sets in the vermicelli elements and whether they are starred or not.*

Proof. For the former claim, note that during reading a finite input word, the fourth component keeps track of in which state a run in $\mathcal{B}$ could be in. Only if there is no accepting state in which $\mathcal{B}$ could be

Algorithm 1 Translating a DFA $\mathcal{A}$ into a non-deterministic very-weak automaton $\mathcal{B}$.

1: $\mathcal{B} = (\varnothing, \Sigma, \varnothing, \varnothing, \varnothing)$
2: **repeat**
3: $\quad\mathcal{D}$ = vermicelli searching automaton for $\mathcal{A}$ and $\mathcal{B}$
4: $\quad r$ = accepting run of minimal distance in $\mathcal{D}$
5: $\quad$**if** r was found **then**
6: $\quad\quad$Add r as vermicelli to $\mathcal{B}$
7: $\quad$**end if**
8: **until** $\mathcal{L}(\mathcal{D}) = \varnothing$

in, $\mathcal{D}$ can accept a word. At the same time, for any word in the language of $\mathcal{A}$, we can always take transitions that reset the Boolean flag to **false**, and set the third component to the set that contains only the current alphabet symbol. This way, the second component only contains a singleton state in every state along the path and represents a path to the accepting state in $\mathcal{A}$.

For the second claim, given a word that is not accepted by $\mathcal{B}$ and a vermicelli that contains the word, the construction of $\mathcal{D}$ ensures that we can build an accepting run on which the last component of the state tuples along the run always represents in which states in $\mathcal{B}$ we can be after reading the respective prefix word. As the word is not accepted by $\mathcal{B}$, the set of these states does not intersect with $F^{\mathcal{B}}$. During the run, the second and third components of the states of $\mathcal{D}$ are used to represent the vermicelli elements, and are updated whenever for the word chosen, we move to the next vermicelli element. The Boolean flag represents whether the current factor in the vermicelli is starred or not, and the second component represents the subset of the alphabet in the vermicelli element. Since a transition has a cost of 1 precisely at the points at which we change these elements (and we need to change these elements in the first transition in $\mathcal{D}$'s run), the run has a cost of k. Along the way, the first component is updated to also represent in which states $\mathcal{A}$ can be after having read any of the words in the language of the vermicelli so far. Since by assumption, we are concerned with vermicelli whose models are contained in the language of $\mathcal{A}$, all of these runs need to be in an accepting state at the end of the run. Overall, the run outlined here is accepting. □

As a consequence, since every NVWF of size n can be described by a set of vermicelli in which each vermicelli is of length at most n, we can compute a NVWF representation of $\mathcal{A}$ using Algorithm 1. Note that the algorithm does not terminate if $\mathcal{A}$ cannot be represented as a very-weak automaton. Since we can however apply the algorithm by Bojańczyk (2008) beforehand to verify the translatability, this imposes no problem. Also note that algorithm 1 can compute a canonical NVWF for a given language by adding lexicographical minimality as a secondary search criterion apart from least distance for finding the next vermicelli in line 4 of the algorithm. By preferring lexicographically smaller vermicelli, it can be ensured that regardless of the syntactic structure of the DFA that we start with, the resulting NVWF will always be the same (as it is built from the same vermicelli). This result can be lifted to the UVW case as the end-component decomposition that we build from a deterministic automaton over infinite words in this case is also unique.

13.3 ACTL ∩ LTL synthesis workflow

Using the translation procedure from the common fragment of ACTL and LTL to universal very-weak automata, we now describe how it can be applied in an overall synthesis workflow.

13.3.1 Building UVWs for the assumptions and guarantees

The first step in the synthesis approach is to translate all properties (i.e., the assumptions and guarantees) to universal very-weak automata.

For maximum efficiency, it is advisable to use an LTL-to-Büchi translator on the negation of a property to see if the result happens to be very-weak. If this is the case, then we already have a UVW for the property under concern and can skip the more complicated translation procedure outlined in the previous section. In practice, this happens very often. For example, for all basic safety properties that can be used as assumptions and guarantees in GR(1) synthesis without pre-synthesis, an LTL-to-Büchi

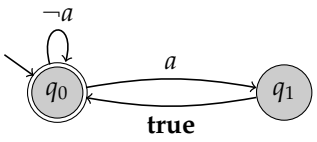

Figure 13.9: A deterministic Büchi automaton for the language GFa that shows that quotienting-based automaton minimisation is not sound for using language equivalence as equivalence relation. In the automaton, all states have the same language, but there exists no one-state automaton for the same language, so a quotienting-based approach would produce an incorrect output automaton.

translator typically yields UVWs right away. Whenever the translator does not find a UVW, we need to apply the procedure from the previous section.

We explained in Chapter 12.2.1 how to obtain a deterministic Büchi automaton from an LTL property whenever possible. As we know that if there exists a UVW for a language, then there exists a deterministic Büchi automaton (Proposition 6), we can assume without loss of generality that the translation succeeds, as otherwise our property under concern is not in ACTL ∩ LTL.

Then, it makes sense to minimise the Büchi automaton before applying the procedure from the previous section. We have seen in Chapter 12.2.1 how to do so (for the more general case of one-pair Rabin automata).

13.3.2 Minimising and merging the UVWs

After we have translated all individual LTL assumptions and guarantees to UVWs, the assumption UVWs can be merged to one big UVW by taking the conjunction of all assumption UVWs. For the guarantees, we do the same. This has the advantage that in the later steps of the synthesis procedure, we can assume that we are dealing with only one assumption UVW and one guarantee UVW.

In order to obtain the best possible performance, we reduce the sizes of the UVWs as much as possible before building the synthesis game from them. For non-deterministic Büchi automata, the existence of a smaller equivalent automaton is a PSPACE-complete problem (Etessami and Holzmann, 2000; Fritz, 2003). Due to the high complexity of the minimisation process, such an operation is typically done in a heuristic manner, i.e., we apply a minimisation procedure that is not guaranteed to yield an automaton of minimal size. For UVWs, it can be shown that the problem has the same complexity, and thus, it is reasonable to minimise these heuristically as well.

For non-deterministic Büchi automata, most minimisation techniques are based on *quotienting*, where we compute an equivalence relation over the states in the automaton and then build a quotient automaton in which all states connected by the relation are merged to one respective state. As equivalence relation, we compute a bisimulation relation over the states in the automaton that guarantees that the quotient automaton under the relation has the same language (Etessami et al., 2001; Clemente, 2011). Suitable bisimulation relations underapproximate language-equivalence of the states in the Büchi automaton, as the language-equivalence relation is not sound under quotienting (Figure 13.9 shows why) and computing the precise language-equivalence relation is a problem with high complexity. As an alternative approach to quotienting, trial-and-error state merging has been proposed (Gurumurthy et al., 2002), which can work with more general bisimulation relations.

For minimising universal very-weak automata, we can apply similar ideas, but must ensure that the automaton stays very-weak. Quotienting in its classical form, where we let those transitions be in the quotient for which we have some transition between states in the represented equivalence class, does not guarantee this (Figure 13.10 shows why). However, this imposes no problem, as we only need to replace quotienting by a construction that merges states in a slightly smarter way. Then, very-weakness is actually a chance, as it always allows to remove some state in the automaton whenever there are two language-equivalent non-rejecting states, and actually allows for even more complex optimisations that are not sound in the non-deterministic Büchi automaton case. Additionally, very-weakness allows us to compute simulation relations in a faster way.

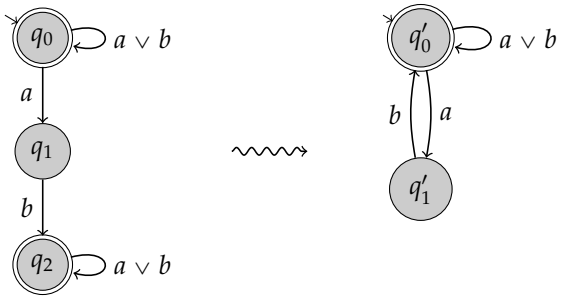

Figure 13.10: A UVW for language $\Sigma^\omega \backslash (a \vee b)^\omega$ that shows that classical quotienting constructions do not preserve very-weakness (on the left). The states q_0 and q_2 have the same language, and can be merged. However, standard quotienting would result in the automaton on the right-hand side, which is not very-weak any more.

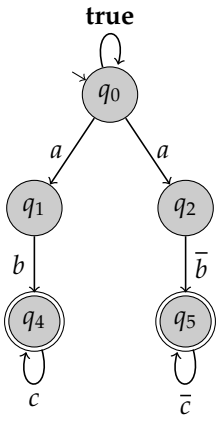

Figure 13.11: A UVW in which the states q_1 and q_2 are always reached at the same time.

Merging states with the same prefix language Consider the automaton in Figure 13.11. The states q_1 and q_2 have the same prefix language, where we define the prefix language of a state to consist of all finite words under which the state can be reached from an initial state.

As q_1 and q_2 are both non-rejecting, we can merge them. We duplicate all outgoing edges of q_2, attach them to q_1, and finally remove state q_2. The resulting automaton is still very-weak. Furthermore, as state q_2 does not have any effect on the set of paths along which q_1 is reached (as there is no connection from q_2 to q_1 in the automaton), the set of prefix words for q_1 (and all other states except for q_2) remains the same. We can apply this principle in a sound way for every such pair of states to obtain a smaller automaton. When doing so, we must however ensure that the resulting automaton is still very-weak. Thus, when merging some state q with some state q', if there exists some path from q to q' in the automaton, then we have to merge q' into q and not the other way around.

The requirement that both states must not be accepting is crucial for soundness. Consider for example the automaton in Figure 13.12. The states q_1 and q_2 are equi-reachable, but they cannot be merged as they reject different sets of words. However, if in a pair of states with the same prefix language, only the state that we would merge to is rejecting, or both states are rejecting and have the same self-loop label set, then we can still perform the merge in a sound manner.

To obtain a list of pairs of states that have the same prefix language and are thus candidates for merging, we compute a reachability simulation relation $R_L \subseteq Q \times Q$ of the states Q of the automaton. This computation is performed in an incremental fashion for states of increasing depth, where the depth of a state is the maximal length of a self-loop-free path to this state from an initial state. If for some states q, q' we have $(q, q') \in R_P$, then this means that for all finite words $t \in \Sigma^*$, if q is an element at level $|t|$ of a run tree of the automaton for t, then so is q'. If after the computation of R_P, we have $(q, q') \in R_P$ and $(q', q) \in R_P$, then q and q' have the same prefix language. We initialise R_P with all elements $\{(q, q) \mid q \in Q\}$ and all elements $\{(q, q') \mid q$ and q' are initial states without incoming edges$\}$. Then, for every $(q, q') \in Q^2$,

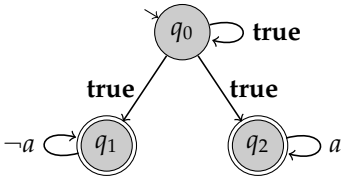

Figure 13.12: A UVW with two states that have the same prefix language, but that cannot be merged without altering the language of the automaton.

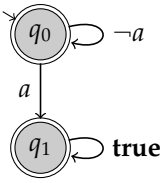

Figure 13.13: A UVW with two states that have the same language.

we add (q, q') to R_P if and only if for every $q'' \in Q$ and $x \in \Sigma$, if $q \in \delta(q'', x)$, then there exists some q''' with $(q'', q''') \in R_P$ such that $q' \in \delta(q''', x)$.

The construction is essentially the same as for computing *direct simulation* over states in a Büchi automaton (see, e.g., Etessami et al., 2001), but we can take advantage of the very-weakness in this case in order to never have to reconsider a pair of states in the computation process of R_P. The computation can be performed in time $|Q|^2 \cdot |\Sigma| \cdot m^2$, where m is the maximum number of incoming edges per state in the UVW.

Merging states with the same language Merging states with the same language is an important step in any optimising LTL-to-Büchi translator. Consider the automaton in Figure 13.13. Both states have the same language and can be merged. For this, we declare one state to be the target state and one state to be the source state, where the depth of the target state must not be lower than the depth of the source state. All incoming transitions to the source state are then rerouted to the target state, and the source state is removed.

As in the case above, we build a simulation relation R_L between the states. For some (q, q') to be in R_L we then need to have that from q, we reject the words that are also rejected from q'. We initialise R_L with all pairs (q, q') such that both q' and q are rejecting states, q' has no non-self-looping outgoing edge, and the self-loop label set of q contains all self-loop labels of q'. Then, we complete R_L from bottom to top, i.e., perform a sorting over $Q \times Q$ with respect to pairwise comparison of the depth of the states, and for every $(q, q') \in Q$, we add (q, q') to R_L if for all $x \in \Sigma$ and $q''' \in Q$ with $q''' \in \delta(q', x)$, we have that there exists some $q'' \in Q$ with $q'' \in \delta(q, x)$ such that one of the following two conditions holds:

- $(q'', q''') \in R_L$, or
- we have $q = q''$, $q' = q'''$, and q is rejecting.

If for some pair (q, q') of states, we have $(q, q') \in R_L$ and $(q', q) \in R_L$, then the states are bisimilar and can be merged. Unlike in the case of the prefix language, there is no additional requirement on whether the states are rejecting or not.

The computation time needed for computing R_L and merging the states is the same as for the prefix-language case. Evaluating R_L corresponds to computing *fair simulation* (Etessami et al., 2001) for the special case of very-weak automata.

Removing superfluous states Sometimes, a state is superfluous as it is dominated by some other state in both the set of words under which it can be reached, as well as by the set of words that are rejected from that state. Such an example is given in Figure 13.14. State q_1 is reached by fewer words than state q_2, but at the same time, from q_2, we reject more words than from q_1. As q_1 and q_2 are not connected in any way, this means that we can simply remove q_1 and all its incoming edges without changing the language of the automaton. Note that then, q_3 becomes unreachable and can also be removed.

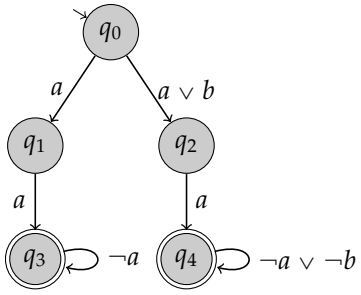

Figure 13.14: A UVW with a superfluous state.

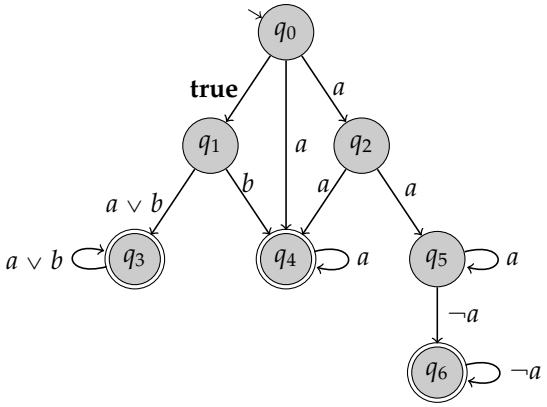

Figure 13.15: A UVW for which it is advisable to merge states with the same language and then merge states with the same set of prefix words. When searching for language-equivalent states, it is concluded that q_1 and q_3 are language-equivalent. State q_1 is then removed and the **true**-edge from q_0 is rerouted to q_3. Then, q_4 loses its incoming b-edge, and only then, q_2 and q_4 are deemed equi-reachable by the simulation relation R_P and can then be merged.

To detect such states, we can reuse the simulation relations that we computed for merging states with the same prefix languages and for merging states with the same languages. If we have $(q, q') \in R_P$ and $(q', q) \in R_L$ for some states q and q', then state q is superfluous.

Repetitive application We have discussed three methods for state space reduction of UVWs above. Typically, we would want to only apply each of them once to obtain a UVW that cannot be reduced by the methods further. Unfortunately, this is not feasible, as regardless of the order of applying these methods, there may always be room for optimisation with these methods left.

Figures 13.15 and 13.16 show two example UVWs for which for the first one, we need to merge states with the same language before two other states become equi-reachable and can subsequently be merged, whereas for the second one, the order is reversed: here, we must first merge some equi-reachable states before an opportunity for state space reduction by merging states with the same language opens.

To reduce the size of the automaton as much as possible, we thus apply all three reduction steps until no more improvement is possible.

13.3.3 Building the synthesis parity game

At this point, we have discussed how to obtain a single UVW representing all the assumptions and a single UVW that represents all the guarantees. To finalise the synthesis approach, we now need to describe how to translate these UVWs into a synthesis game that ensures that precisely the strategies that are winning for the system player are the implementations that satisfy the specification. Even more, the structure of the parity game should be simple enough to allow the efficient usage of symbolic parity game solving algorithms.

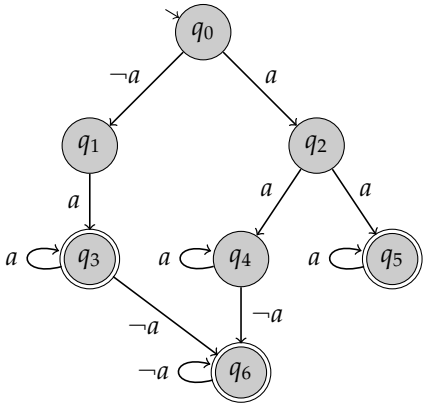

Figure 13.16: A UVW for which it is advisable to merge states with the same set of prefix words before merging states with the same language. When searching for states with the same prefix language, it is concluded that q_4 and q_5 can be merged. State q_5 is then removed and q_4 is made accepting. Only then, q_3 and q_4 have the same language and can subsequently be merged. Finally, q_1 and q_2 are merged due to language equivalence.

For a UVW to accept a word, there must not be any branch in its run tree that eventually enters a rejecting state and never leaves it. During the play of the parity game, we can test this by keeping track of in which states in the UVW a branch of a run tree for the prefix trace can be in at after reading the prefix trace. We then cycle through all of the rejecting states and wait until the state has been left. We make sure that after we cycled through all of the states correctly, the resulting position has colour 1 (for the assumption UVW) or colour 2 (for the guarantee UVW). The resulting parity game construction is similar to the one in Definition 46.

Definition 57. *Let $\mathcal{A}^A = (Q^A, \Sigma, \delta^A, Q^A_{init}, \mathcal{F}^A)$ be an assumption UVW, $\mathcal{A}^G = (Q^G, \Sigma, \delta^G, Q^G_{init}, \mathcal{F}^G)$ be a guarantee UVW, and $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ be an interface such that $\Sigma = 2^{\mathsf{AP}^I} \times 2^{\mathsf{AP}^O}$. Without loss of generality, let $\{q_1, \ldots, q_m\}$ be the rejecting states in $\mathcal{A}^A$ and $\{q'_1, \ldots, q'_m\}$ be the rejecting states in $\mathcal{A}^G$. We define the synthesis game $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{init}, c)$ over $\mathcal{A}^A$, $\mathcal{A}^G$, and $\mathcal{I}$ as follows:*

$$
\begin{aligned}
V^0 &= 2^{Q^A} \times 2^{Q^G} \times \{0, \ldots, m\} \times \{0, \ldots, n\} \\
V^1 &= V^0 \times \Sigma^0 \\
\Sigma^0 &= 2^{\mathsf{AP}^I} \\
\Sigma^1 &= 2^{\mathsf{AP}^O} \\
E^0(v, x) &= (v, x) \text{ for all } v \in V^0, x \in \Sigma^0 \\
E^1((S^A, S^G, i, j, x), y) &= (S'^A, S'^G, i', j') \text{ s.t.} \\
&\quad S'^A = \{q \in Q^A \mid \exists q' \in S^A : q \in \delta^A(q', (x, y))\} \\
&\quad S'^G = \{q \in Q^G \mid \exists q' \in S^G : q \in \delta^G(q', (x, y))\} \\
&\quad i' = \begin{cases} (i+1) \bmod m & \text{if } i = 0 \vee q_i \notin S^A \vee q^i \notin \delta^A(q^i, (x, y)) \\ i & \text{else} \end{cases} \\
&\quad j' = \begin{cases} (j+1) \bmod m & \text{if } j = 0 \vee q_j \notin S^G \vee q^j \notin \delta^G(q^j, (x, y)) \\ j & \text{else} \end{cases} \\
&\quad \text{for all } (S^A, S^G, i, j, x) \in V^1 \text{ and } y \in \Sigma^1 \\
c(v) &= \begin{cases} 2 & \text{if } v \in V^0 \text{ and } v = (S^A, S^G, i, 0, x) \text{ for some } S^A, S^G, i, x \\ 1 & \text{if } v \in V^0 \text{ and } v = (S^A, S^G, 0, j, x) \text{ for some } S^A, S^G, j \neq 0, x \\ 0 & \text{otherwise} \end{cases} \\
V^0 &= (Q^A_{init}, Q^G_{init}, 0, 0)
\end{aligned}
$$

Theorem 15. *Let $\mathcal{A}^A = (Q^A, \Sigma, \delta, Q^A_{init}, \mathcal{F}^A)$ be an assumption UVW, $\mathcal{A}^G = (Q^G, \Sigma, \delta, Q^G_{init}, \mathcal{F}^G)$ be a guarantee*

UVW, and $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ *be an interface such that* $\Sigma = 2^{\mathsf{AP}^I} \times 2^{\mathsf{AP}^O}$*. Precisely the strategies that are winning for player* 1 *(who tries to ensure that the highest colour occurring infinitely often in a play is even) in a game built from* $\mathcal{A}^A$*,* $\mathcal{A}^G$ *and* $\mathcal{I}$ *according to Definition 57 are the ones that represent Mealy-type implementations whose runs are in the language of* $\mathcal{A}^G$ *whenever they are in the language of* $\mathcal{A}^A$*.*

Proof. The correctness follows from the fact that the construction essentially consists of two implementations of the UVW-to-DBW construction from Proposition 6, and we visit colour 1 whenever the assumption DBW has just visited an accepting state, and colour 2 whenever the guarantee DBW just visited an accepting state. In case both events occur at the same time, we only visit colour 2. This way, all traces are winning that satisfy the guarantee UVW or that do not satisfy the assumption UVW. Taking into account that the construction spreads the overall alphabet in a Mealy-type manner, the claim follows. □

13.3.4 Solving the game and computing an implementation

After the synthesis parity game has been built (and ideally, is represented in a symbolic manner), we can apply any off-the-shelf parity game solving algorithm to check the realisability of the game. Using binary decision diagrams (BDDs) as reasoning backend, the fixed-point based characterisation from Chapter 12.5 is suitable as a basis for such an algorithm.

After the solution of the game, whenever we find it to be winning for the system player, we first compute a *general strategy* that follows the prefixed points as described in Chapter 12.5. In some cases, there may be multiple possible moves by the system player that all lead to successor positions in the same prefixed point of winning positions. This motivates the term *general strategy* in this context, as the strategy does not fix which of these moves to take. As every specialisation of this strategy is winning, we can thus use the freedom for optimising the size of the strategy's implementation as a circuit.

One method to do is that is easy to use has been proposed by Kukula and Shiple (2000). Alternatively, a more advanced computational-learning-based circuit extraction scheme (Ehlers et al., 2012a) that tries to make most efficient use of the generality of the strategy can be applied.

13.4 Conclusion

In this chapter, we have presented the first approach for synthesis from the common fragment of ACTL and LTL. The beauty of the approach is that it allows us to harness the simplicity and reasoning efficiency of universal very-weak automata over infinite words. The key ingredient that was missing in the literature in order to make such an approach feasible was the procedure to translate an LTL formula into this automaton form, whenever possible. This lack has been fixed in this work, making the specification fragment attractive for future synthesis endeavours.

As a side-product, we have solved a long-standing open problem on translating between ACTL and LTL. Clarke and Draghicescu (1988) showed that if a CTL formula has an equivalent LTL formula, then the latter can be obtained by removing all path quantifiers from the CTL formula. This fact is often cited in lectures and textbooks on verification. Previously, it was unknown how to reverse the direction of the translation. Our new LTL-to-UVW translation procedure provides an answer. Since Maidl (2000, Lemma 11) described a procedure to translate a UVW to an equivalent ACTL formula, we obtain as a corollary a procedure to translate from LTL to ACTL, whenever possible. Note that we cannot however get rid of the "A" in ACTL here. Bojańczyk (2008) described a language that is representable in LTL and CTL, but not in ACTL. Such pathological cases are not covered by our procedure. However, for all practical means, the translation is now known.

Chapter 14

On the relationship of GRabin(1) and ACTL ∩ LTL synthesis

In the previous chapters, we introduced two new techniques for the efficient synthesis of reactive systems from formal specifications. To simplify presentation, we have refrained from discussing any ideas that are common to both approaches or are concerned with their combination. However, these ideas are not ignored, and we discuss them in this chapter.

We first examine how to connect the two approaches to a synthesis workflow that (1) can handle both liveness and stability properties, (2) is based on very-weak automata for representing the assumptions and guarantees and (3) reduces the synthesis problem to parity game solving. Our main interest in such an approach is its application to robust synthesis, for which we subsequently refine the workflow in Chapter 14.2.

Then, we introduce a way to get rid of some round-robin counters in the parity games that are built when combining GRabin(1) synthesis and ACTL ∩ LTL synthesis. We base the technique on a new automaton model, *globalised co-Büchi automata*, which we will show to have favourable properties for the synthesis of robust systems.

14.1 Combining GRabin(1) and ACTL ∩ LTL synthesis

The advantages of GRabin(1) and ACTL ∩ LTL synthesis over the GR(1) synthesis approach are in some way orthogonal: the former approach extends the expressiveness of GR(1) synthesis by introducing support for stability properties, and thus allows to synthesise robust systems, whereas the latter technique introduces a way to get rid of pre-synthesis in a natural way.

The question of how to combine these advantages is thus natural. In GR(1) synthesis, all properties can be used that are representable as deterministic Büchi word automata (after pre-synthesis). For GRabin(1) synthesis, this property set is extended to those representable as deterministic one-pair Rabin automata, which have a conjunction of a Büchi acceptance condition and a co-Büchi acceptance condition. Thus, by the transition from GR(1) to GRabin(1) synthesis, we add the possibility to use properties as assumptions and guarantees for which previously only their complement could be used.

For ACTL ∩ LTL synthesis, a similar transition to *(ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis* is desirable, as it would enrich the set of properties that can be used by stability properties, and allows the application of the ideas for robust system synthesis presented in Chapter 12.

The complement of the languages representable as universal very-weak word automata are those representable as non-deterministic very-weak automata over infinite words (NVWs), thus when transitioning from ACTL ∩ LTL synthesis to (ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis, we add support for having these automata as assumptions and guarantees. Unlike UVWs, NVWs can however blow up when taking their conjunction. Thus, when we have many of these as assumptions and guarantees, taking the conjunction of all assumption NVWs and guarantee NVWs as we did for the assumption and guarantee UVWs in ACTL ∩ LTL synthesis seems not to be a promising approach. However, we will see later that for the majority of properties that we are dealing with, including the ones that are introduced by the specification ruggedisation process for robust synthesis, we can replace the NVWs by *globalised co-Büchi automata*. For checking whether some word is accepted by an automaton of this class, we observe if finitely many prefixes of a word are rejected by the automaton. Globalised co-Büchi automata do not blow up under conjunction and recover our ability to incorporate a large number of assumptions and guarantees of all supported types in an efficient synthesis workflow.

To translate all properties into UVWs and NVWs, we start as in ACTL ∩ LTL synthesis. We check if a property under concern is identified as being very-weak by a simple call to an LTL-to-Büchi translator,

and are done with the property in that case. Otherwise, we compute a one-pair Rabin automaton as described in Chapter 12.2.1 and split it into a deterministic Büchi word automaton and a deterministic co-Büchi word automaton that both need to accept for the Rabin automaton to accept a word. The Büchi automaton can then be translated to a UVW as described in the previous chapter, and the co-Büchi automaton can be translated to a NVW by again applying the construction from the previous chapter to the co-Büchi automaton, as by duality, it is also suitable for this purpose.

In some rare cases, this process might fail despite the fact that the language of the one-pair Rabin automaton is representable as the intersection of the language of a NVW and the language of a UVW. An example is a Rabin automaton accepting the empty language for which its Büchi acceptance condition part accepts precisely the words in $\Sigma^* a a \Sigma^*$ (for some a in the alphabet Σ), and the co-Büchi part rejects precisely the words in $\Sigma^* a a \Sigma^*$. We have seen in the proof of Proposition 3 that these languages are not representable as UVWs and NVWs, respectively. Yet, as the language of the Rabin automaton is empty, its representation as the intersection of the languages of a UVW and a NFW is possible. By taking such properties "on the side" in the constructions to follow (i.e., keeping the deterministic Büchi and co-Büchi automata obtained from the deterministic one-pair Rabin automaton explicit) and encode them as a product in the synthesis game that we build in the following, we can still handle such (rare) cases.

After all UVWs and NVWs are computed, we can then build a parity game from them that combines the ideas of Definition 46 and Definition 57. To keep the length of the description of the following construction reasonable, we do not define the game in one step, but rather first translate the UVWs to deterministic Büchi automata and the NVWs to deterministic co-Büchi automata. In both cases, the construction from Proposition 6 is useful. The state spaces of these DBWs and DCWs are then factors of the position space of the parity game, and the DBWs and DCWs come with a suitable symbolic encoding, as their state spaces are the products of a counter and a power set over the states of the underlying very-weak automaton.

Definition 58. *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be a set of deterministic co-Büchi word automata with $\mathcal{A}_i = (Q_i, \Sigma, q_{init,i}, \delta_i, \mathcal{F}_i)$ for all $i \in \{1, \ldots, n\}$. We define its conjunction to be the deterministic co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, q_{init}, \delta, \mathcal{F})$ with:*

$$\begin{aligned} Q &= Q_1 \times \ldots \times Q_n \\ q_{init} &= (q_{init,1}, \ldots, q_{init,n}) \\ \delta((q_1, \ldots, q_n), x) &= (\delta_1(q_1, x), \ldots, \delta_n(q_n, x)) \text{ for all } (q_1, \ldots, q_n) \in Q, x \in \Sigma \\ \mathcal{F} &= \{(q_1, \ldots, q_n) \in Q \mid q_1 \in \mathcal{F}_1 \vee \ldots \vee q_n \in \mathcal{F}_n\} \end{aligned}$$

Lemma 17. *The construction in Definition 58 computes for a given set of DCWs a new DCW that represents the intersection of the languages of the individual DCWs.*

Proof. The construction essentially simulates all automata running in parallel. Note that a word is in the language all of the DCWs if none of them visits rejecting states infinitely often. This is correctly simulated by letting all states be rejecting in the product for which any of the automata is in a rejecting state. □

As Definition 58 describes how to take the conjunction of the assumption DCWs and guarantee DCWs, we can merge the assumption and guarantee UVWs prior to translating them to DBWs, and we can merge the assumption and guarantee NVWs after translating them to DCWs, we can assume in our parity game construction that we have precisely one automaton for every combination of assumption/guarantee and co-Büchi/Büchi acceptance.

Definition 59. *Let $\mathcal{A}^{BA} = (Q^{BA}, \Sigma, q^{BA}_{init}, \delta^{BA}, \mathcal{F}^{BA})$ be an assumption DBW, $\mathcal{A}^{BG} = (Q^{BG}, \Sigma, q^{BG}_{init}, \delta^{BG}, \mathcal{F}^{BG})$ be a guarantee DBW, $\mathcal{A}^{CA} = (Q^{CA}, \Sigma, q^{CA}_{init}, \delta^{CA}, \mathcal{F}^{CA})$ be an assumption DCW, $\mathcal{A}^{CG} = (Q^{CG}, \Sigma, q^{CG}_{init}, \delta^{CG}, \mathcal{F}^{CG})$ be a guarantee DCW, and $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ be an interface such that $\Sigma = 2^{\mathsf{AP}^I} \times 2^{\mathsf{AP}^O}$. Without loss of generality, we assume that for all of these automata $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$, we have $|\delta(q, x)| = 1$ for all $q \in Q$ and $x \in 2^{\mathsf{AP}}$ (i.e., there are no dead-ends in the automaton). We define the synthesis parity game for this setting as $\mathcal{G} = (V^0, V^1, \Sigma^0, \Sigma^1, E^0, E^1, v_{init}, c)$ with:*

- $V^0 = Q^{BA} \times Q^{BG} \times Q^{CA} \times Q^{CG} \times \mathbb{B}$
- $V^1 = V^0 \times \Sigma^0$

- $\Sigma^0 = 2^{\mathsf{AP}^I}$
- $\Sigma^1 = 2^{\mathsf{AP}^O}$
- *For all* $(q^{BA}, q^{BG}, q^{CA}, q^{CG}, b) \in V^0$ *and* $x \in \Sigma^0$, $E^0((q^{BA}, q^{BG}, q^{CA}, q^{CG}, b), x) = (q^{BA}, q^{BG}, q^{CA}, q^{CG}, b, x)$
- *For all* $(q^{BA}, q^{BG}, q^{CA}, q^{CG}, b, x) \in V^1$ *and* $y \in \Sigma^1$, *we have* $E^1((q^{BA}, q^{BG}, q^{CA}, q^{CG}, b, x), y) = (\delta^{BA}(q^{BA}, (x, y)), \delta^{BD}(q^{BG}, (x, y)), \delta^{CA}(q^{CA}, (x, y)), \delta^{CG}(q^{CG}, (x, y)), b')$, *where* $b' =$ **true** *if and only if*
  (1) $q^{BA} \in \mathcal{F}^{BA}$, *or*
  (2) $b =$ **true** *and* $q^{CG} \notin \mathcal{F}^{CG}$.
- *c maps all vertices of player* 1 *to* 0 *and all vertices* $v = (q^{BA}, q^{BG}, q^{CA}, q^{CG}, b)$ *of player* 0 *to the least number in* $\{0, 1, 2, 3, 4\}$ *such that:*
  - $c(v) = 4$ *if* $q^{CA} \in \mathcal{F}^{CA}$
  - $c(v) \geqslant 3$ *if* $q^{CG} \in \mathcal{F}^{CG}$ *and* $b =$ **true**
  - $c(v) \geqslant 2$ *if* $q^{BG} \in \mathcal{F}^{BG}$
  - $c(v) \geqslant 1$ *if* $q^{BA} \in \mathcal{F}^{BA}$

Theorem 16. *Let* $\mathcal{A}^{BA} = (Q^{BA}, \Sigma, q^{BA}_{init}, \delta^{BA}, \mathcal{F}^{BA})$ *be an assumption DBW,* $\mathcal{A}^{BG} = (Q^{BG}, \Sigma, q^{BG}_{init}, \delta^{BG}, \mathcal{F}^{BG})$ *be a guarantee DBW,* $\mathcal{A}^{CA} = (Q^{CA}, \Sigma, q^{CA}_{init}, \delta^{CA}, \mathcal{F}^{CA})$ *be an assumption DCW,* $\mathcal{A}^{CG} = (Q^{CG}, \Sigma, q^{CG}_{init}, \delta^{CG}, \mathcal{F}^{CG})$ *be a guarantee DCW, and* $\mathcal{I} = (\mathsf{AP}^I, \mathsf{AP}^O)$ *be an interface such that* $\Sigma = 2^{\mathsf{AP}^I} \times 2^{\mathsf{AP}^O}$. *Precisely the winning strategies of the system player in a parity game built from* $\mathcal{I}$, $\mathcal{A}^{BA}$, $\mathcal{A}^{BG}$, $\mathcal{A}^{CA}$, *and* $\mathcal{A}^{CG}$ *according to Definition 59 are the ones that ensure that along any play of the game, either* $\mathcal{A}^{BA}$ *does not accept the corresponding decision sequences,* $\mathcal{A}^{CA}$ *does not accept the decision sequences, or* $\mathcal{A}^{BG}$ *and* $\mathcal{A}^{CG}$ *accept the decision sequences.*

Proof. The proof can be performed in the same way as for Theorem 12, i.e., by a case distinction over the possible combinations of acceptance/non-acceptance of the four automata. The case here is actually a bit simpler, as there is no need to track if safety assumption or guarantee violations have occurred in the past – the respective automata keep track of this themselves. □

14.2 Robust synthesis

The construction for (ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis from the previous section requires the translation of all NVWs that we computed from the properties that are not representable as DBWs, but only as DCWs, to the latter automaton type. In every translation step from a NVW to such a deterministic co-Büchi automaton, a counter is introduced. If we ruggedise a specification from practice, which typically contains many safety properties, then we obtain a lot of co-Büchi automata in this way. We then need to compute a new deterministic co-Büchi automaton for their conjunction (using Definition 58) and can then use the resulting automaton in the synthesis game from Definition 59. Introducing counters in all of the co-Büchi automata leads to an automaton for their conjunction that encodes many counters. This blows up the position space of the resulting synthesis game, and thus should be avoided if possible.

Furthermore, the co-Büchi automata might have some undesirable properties. In particular, when we are concerned with ruggedised specifications, but want to ensure that the synthesised implementation satisfies all safety guarantees until a temporary assumption violation has been witnessed (as in Chapter 12.5.1), the co-Büchi automaton for a ruggedised version of a safety property has to visit a rejecting state as soon as such a temporary property violation for the original safety property is witnessed, and not earlier. Only then, we can make use of the fact that in a fixed-point based game solving algorithm, satisfying the requirement that the synthesised implementation can only temporarily violate safety guarantees after a temporary assumption violation has been witnessed comes for free. We call such automata *co-Büchi tight* for the scope of this chapter.

Luckily, for the scope of robust synthesis, we can save a lot counters and at the same time ensure the co-Büchi tightness of our DCWs by exchanging only one part of the synthesis construction. The key idea is to use a different intermediate automaton model for the DCW-type properties that allows us to get rid of **all** the counters when translating from the model to a symbolically representable deterministic co-Büchi automaton. At the same time, the automaton model lends itself to handling ruggedised specifications. We define this model as follows:

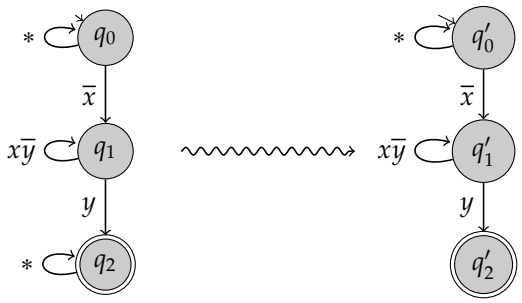

Figure 14.1: On the left-hand-side: a UVW for the LTL specification $\mathsf{G}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$. On the right-hand-side: a GlCA for the LTL specification $\mathsf{FG}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$ that originates from ruggedizing the former specification.

Definition 60. *Let Σ be an alphabet. We define a universal globalised co-Büchi automaton as a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ with the set of states Q, the alphabet Σ, the transition function $\delta : Q \times \Sigma \rightarrow 2^Q$, the initial states $Q_0 \subseteq Q$, and the set of final states $\mathcal{F}$.*

Given a word $w = w_0 w_1 \ldots \in \Sigma^\omega$, we say that some sequence $\pi = \pi_0 \pi_1 \ldots \pi_n$ is a run of $\mathcal{A}$ over w if $\pi_0 \in Q_0$ and for all $0 < i < n$, $\pi_{i+1} \in \delta(\pi_i, w_i)$. The run π is called rejecting if and only if $\pi_n \in \mathcal{F}$.

We say that $\mathcal{A}$ rejects a word w if for infinitely many values n, there exists a rejecting run of length n for w. All other words are accepted by the automaton.

Universal globalised co-Büchi automata (*GlCA*) are similar to ordinary non-deterministic co-Büchi automata, with the exception that the infinitely many visits to rejecting states that are required by the co-Büchi acceptance condition for rejecting a word may occur along different runs of the globalised automaton. Thus, in a sense, the main difference is that during the run of the automaton, we may jump between the possible states that we can be in after reading a prefix word.

To illustrate the idea of globalised automata, consider a universal very-weak automaton for the specification $\mathsf{G}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$ over the atomic proposition set $\{x, y\}$. The automaton is depicted on the left-hand side of Figure 14.1. One can immediately see from the fact that there is only one rejecting state with a self-loop for every alphabet symbol that the automaton represents a safety language. The right-hand side of the figure shows a GlCA for the ruggedisation of the same specification. Essentially, for translating between the two models in this case, it suffices to remove/add the unconditional self-loop for the rejecting state. The rejecting state can be visited along a run whenever we have just witnessed in a word that $\neg x \rightarrow \mathsf{X}\neg(x\,\mathsf{U}\,y)$ does not hold at position i of the word, and precisely for these indices i, there is a run of the same length that ends in the rejecting state. As for the automaton to reject a word, there have to be infinitely many values $i \in \mathbb{N}$ such that there exists a rejecting run of that length for the word, the GlCA represents the specification $\mathsf{FG}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$. The beauty of the automaton class is the ease of translating its members to DCWs.

Lemma 18. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a globalised co-Büchi word automaton. The co-Büchi word automaton $\mathcal{A}' = (Q', \Sigma, \delta', q'_0, \mathcal{F}')$ with:*

- $Q' = 2^Q$,
- *For all $S \in Q'$ and $x \in \Sigma$, $\delta'(S, x) = \{q' \in Q \mid \exists q \in S : q' \in \delta(q, x)\}$,*
- $q'_0 = Q_0$, *and*
- $\mathcal{F} = \{S \subseteq Q \mid S \cap F \neq \varnothing\}$

has the same language as $\mathcal{A}$.

Proof. Let $w = w_0 w_1 \ldots \in \Sigma^\omega$ be a word. By the definition of q'_0 and δ', the states of $\mathcal{A}'$ track precisely in which states of $\mathcal{A}$ we can be in after reading a prefix of w.

Assume that w is rejected by $\mathcal{A}$. Then there exist infinitely many indices $i \in \mathbb{N}$ such that there exists a run for w in $\mathcal{A}$ of length i that ends in a state in $\mathcal{F}$. For every such i, the construction ensures that then at the ith position in a run of $\mathcal{A}'$ for w, we are in some state $S \in Q'$ for which for some $q \in \mathcal{F}$, we have $q \in S$. Thus, we are then in a rejecting state of $\mathcal{A}'$. As this happens infinitely often, $\mathcal{A}'$ rejects w.

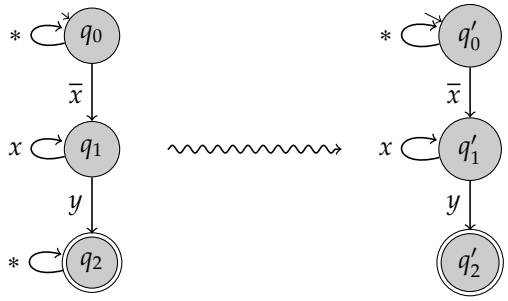

Figure 14.2: Automata for Example 19. On the left-hand-side: a UVW for the LTL specification $\mathsf{G}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$. On the right-hand-side: a GlCA that does not represent the LTL property $\mathsf{FG}(\neg x \rightarrow \mathsf{X}\neg(x\mathsf{U}y))$.

On the other hand, assume that w is rejected by $\mathcal{A}'$. Then, for the unique run of $\mathcal{A}'$ for w, rejecting states of $\mathcal{A}$ are infinitely often in the state label on the run of $\mathcal{A}'$. As $\mathcal{A}'$ tracks in which states of $\mathcal{A}$ we can be in after reading some prefix word, whenever this happens, we have found a rejecting run of $\mathcal{A}'$ for that prefix word. As this event occurs infinitely often, $\mathcal{A}$ rejects the word as well. □

Thus, we do not need to introduce any counters when translating a GlCA to a deterministic co-Büchi word automaton, unlike for the translation from non-deterministic very-weak automata. Rather, a simple power-set construction suffices, and it allows us to seamlessly switch focus between different runs of the original GlCA. This approach is also useful in other applications of automata theory. For example, Kupferman and Vardi (2001b) apply the concept to the model checking of safety properties that are given as non-deterministic automaton.

To now apply GlCAs in robust synthesis, we need to know how to obtain a GlCA for a ruggedised property. Doing so is not always as easy as the case of the automata in Figure 14.1 suggests, which the following example shows.

Example 19. Consider the UVW on the left-hand side of Figure 14.2. The UVW is the same as the one from Figure 14.1, except that the self-loop on state q_1 is labelled by x and not $x\overline{y}$. The language of the UVW is not affected by this change. However, the GlCA that we obtain from the UVW by removing the self-loops in the rejecting state, which is depicted on the right-hand side of the figure, does not represent the ruggedisation of the language of the UVW this time (unlike for the automata in Figure 14.1).

To see this, consider the word $w = \varnothing(\{x, y\})^\omega$. As we can stay in q_1 indefinitely long, there are rejecting runs for the GlCA of length n for every $n \geqslant 2$. These runs stay in state q'_1 until they eventually take the transition to q'_2. Thus, the GlCA rejects w although it actually is a model of the LTL property $\mathsf{FG}(\neg x \rightarrow \mathsf{X}\neg(x\,\mathsf{U}\,y))$, which is equivalent to the ruggedisation of the UVW on the left-hand side of Figure 14.2. The reason is that from w^1 onwards, x is always true, so $\mathsf{FG}(\neg x \rightarrow \ldots)$ always holds. Thus, we have an example where a GlCA for a ruggedised specification cannot simply be obtained from a UVW for the non-ruggedised version of the specification by removing self-loops of the rejecting state. This example also suggests that in a sound procedure to compute a GlCA from a specification, we must avoid introducing states such as q'_1, in which runs can be "parked" indefinitely long despite witnessing property violations. ★

The sound construction to obtain a GlCA for a ruggedised property that we present in the following consists of two steps. We start with a lemma on obtaining deterministic automata over finite words that we can use as starting point for the translation.

Lemma 19. *Let ψ be a safety LTL property and $\mathcal{A} = (Q, \Sigma, \delta, q_0, \mathcal{F})$ be a deterministic Büchi word automaton for ψ. Without loss of generality, assume that $\mathcal{A}$ does not have dead-ends, i.e., for all $q \in Q$, $x \in \Sigma$, we have $|\delta(q, x)| = 1$. Let furthermore $Q' \subseteq Q$ be the set of states of $\mathcal{A}$ with the empty language, i.e., every word in Σ^ω is rejected from these states.*

We construct a deterministic automaton over finite words that accepts all words $w = w_0 \ldots w_n \in \Sigma^*$ for which $w_0 \ldots w_n$ is a bad prefix for ψ, but $w_0 \ldots w_{n-1}$ is no such prefix. We do so by removing all outgoing edges from the states in Q', and making precisely the states Q' accepting.

Proof. As $\mathcal{A}$ is deterministic and the set of ψ's models is a safety language, we enter a state with the empty language after having observed a bad prefix. At the same time, if $\mathcal{A}$ is in a state that does not

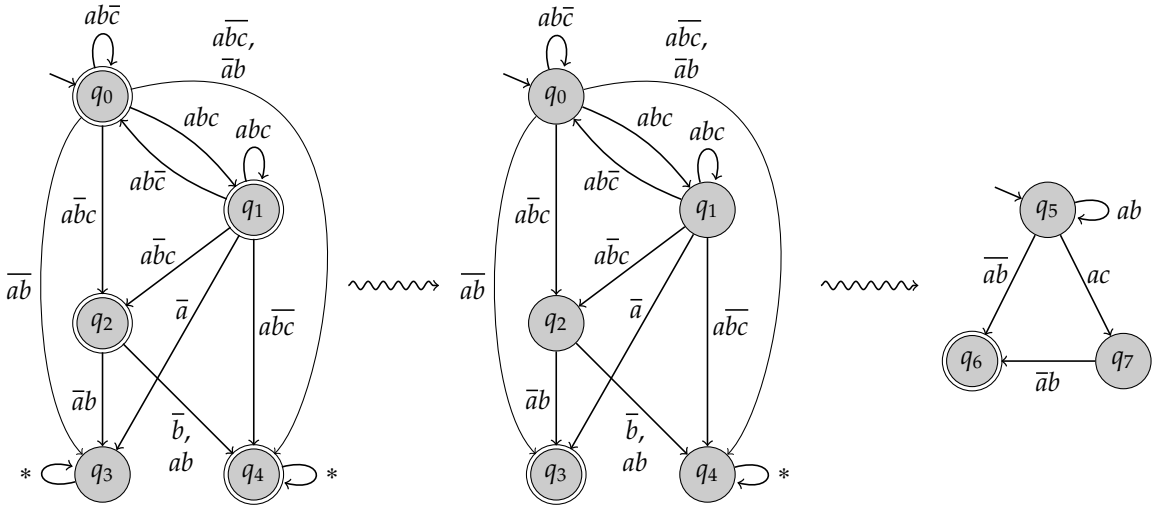

Figure 14.3: A deterministic Büchi automaton for the LTL property $\neg((a \wedge b)\,\mathsf{U}\,(a \wedge c \wedge \mathsf{X}(\neg a \wedge b))) \wedge \neg((a \wedge b)\,\mathsf{U}\,(\neg a \wedge \neg b))$ (on the left-hand side), its prefix-tight automaton that is obtained using the construction from Lemma 19 (in the middle), and a NVWF that accepts the same language as the prefix-tight automaton (on the right-hand side).

have an empty language, then the word read so far cannot be a bad prefix for $\mathsf{G}\psi$. Thus, we enter a state in Q' precisely after observing a prefix word that is a bad prefix for $\mathsf{G}\psi$, but no shorter prefix of the word is also a bad prefix. By making the states in Q' accepting, but removing all outgoing edges from states in Q', we ensure to accept precisely these words. □

Given a deterministic Büchi automaton $\mathcal{A}$ that accepts a safety language, we call the deterministic automaton over finite words that we obtain from $\mathcal{A}$ by applying the construction from Lemma 19 the *prefix-tight automaton* for $\mathcal{A}$.

Figure 14.3 shows an example for the translation of a deterministic Büchi automaton that represents a safety property to a prefix-tight automaton. The figure also gives an example for a subsequent translation of the prefix-tight automaton to a non-deterministic very-weak automaton over finite words by the construction in Chapter 13.2.2, which is a step that we use in the following theorem.

Theorem 17. *Let ψ be a safety LTL property, $\mathcal{A}$ be a deterministic Büchi word automaton for ψ and $\mathcal{A}'$ be the prefix-tight automaton of $\mathcal{A}$. Let furthermore $\mathcal{A}''$ be a NVWF that accepts the same language as $\mathcal{A}'$ (that may have been computed from $\mathcal{A}'$ by the construction from Chapter 13.2.2).*

Claim 1: *We can obtain a GlCA $\mathcal{A}'''$ for the LTL property $\mathsf{FG}\psi$ from $\mathcal{A}''$ by adding self-loops for all initial states and all letters (of the alphabets of $\mathcal{A}$, $\mathcal{A}'$, and $\mathcal{A}''$).*

Claim 2: *Let $w = w_0 w_1 \ldots$ be a word and $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be a partial function that maps every $k \in \mathbb{N}$ to the least number m such that $w_k \ldots w_m$ is a bad prefix for ψ. If no such bad prefix exists, then $f(k)$ is undefined. The GlCA from Claim 1 has the property that precisely for the values of $n \in \mathbb{N}$ such that there exists some $k \in \mathbb{N}$ with $f(k) = n$, there exists a rejecting run for $w_0 \ldots w_n$ for some $n \in \mathbb{N}$.*

Proof. **Claim 1:** Note that the second claim implies the first claim. If there are infinitely many $k \in \mathbb{N}$ such that $f(k)$ is defined, then $\mathsf{FG}\psi$ is not fulfilled as for all of these $k \in \mathbb{N}$, $w^k \not\models \psi$ by the definition of f. Also, for every $k \in \mathbb{N}$, we have $f(k) \geqslant k$ by the definition of f. Thus, if we have infinitely many $k \in \mathbb{N}$ such that $f(k)$ is defined, then we also have infinitely many $n \in \mathbb{N}$ such that $f(k) = n$ for some $k \in \mathbb{N}$ (which can be proven by assuming the converse and deriving a contradiction from the fact that there are infinitely many indices on which f is defined). As for all of these $n \in \mathbb{N}$, $w_0 \ldots w_n$ induces a rejecting word for $\mathcal{A}'''$, the GlCA rejects w.

On the other hand, assume that there are infinitely many $n \in \mathbb{N}$ such that $w_0 \ldots w_n$ has a rejecting run of $\mathcal{A}'''$. For every such n there has to exist some $k \in \mathbb{N}$ such that $w_k \ldots w_n$ is accepted by $\mathcal{A}''$ as $\mathcal{A}'''$ is obtained from $\mathcal{A}''$ by simply adding self-loops in the initial states (and calling the accepting states rejecting). Note that all of these values k have to be different: By the construction of $\mathcal{A}'$, to which $\mathcal{A}''$ is equivalent, we cannot both accept $w_k \ldots w_n$ and $w_k \ldots w_{n'}$ for different values of n and n'. Thus, there

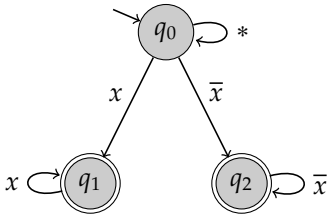

Figure 14.4: A non-deterministic very-weak automaton over infinite words for the LTL property $\mathsf{FG}(x \rightarrow \mathsf{FG}x)$.

exist infinitely many $k \in \mathbb{N}$ such that w^k is accepted by $\mathcal{A}'$, and by the definition of $\mathcal{A}'$, $w_k w_{k+1} \dots$ is rejected by $\mathcal{A}$ and thus $w^k \not\models \psi$. Since we have infinitely many such $k \in \mathbb{N}$, it follows that $w \not\models \mathsf{FG}\psi$.

Claim 2: For proving the second claim, consider a word w with a corresponding function f as defined in the claim. Without loss of generality, we can assume that $\mathcal{A}''$ has only one initial state without a self-loop. We can always ensure this by adding an additional initial state and attaching copies of the outgoing edges for all states that were previously initial to the new initial state. Note that this operation does not affect the language of $\mathcal{A}''$. Also, the language of the GlCA $\mathcal{A}'''$, that we then obtain from the modified version of $\mathcal{A}''$, is unaffected by this operation.

If for some $k \in \mathbb{N}$ and $m \in \mathbb{N}$, we have that $w_k \dots w_m$ is a bad prefix of ψ, then $\mathcal{A}'$ accepts the word $w_k \dots w_m$ by Lemma 19 unless a strict prefix of $w_k \dots w_m$ is also a bad prefix for ψ. However, in the claim, this case is excluded, as we assume m to be the least number such that $w_k \dots w_m$ is a bad prefix. Now since $\mathcal{A}'$ accepts the word, so does $\mathcal{A}''$ (as they are equivalent). Let the accepting run of $w_k \dots w_n$ of $\mathcal{A}''$ start in some state $q \in Q_0$. We obtain a rejecting run of $\mathcal{A}'''$ for $w_0 \dots w_n$ by staying in state q for the first k cycles, and then taking the same path as the run of $\mathcal{A}''$ for $w_k \dots w_n$. As that run ends in an accepting state of $\mathcal{A}''$, the run for $\mathcal{A}'''$ ends in a rejecting state by the definition of the rejecting states for $\mathcal{A}'''$.

On the other hand, assume that for some value of $n \in \mathbb{N}$, there exists no $n \in \mathbb{N}$ such that $f(k) = n$. We show that then, there exists no value of k such that we obtain a rejecting run for $w_0 \dots w_n$ for $\mathcal{A}'''$ by looping k times in the initial state of $\mathcal{A}'''$. As this covers all possible cases for obtaining a rejecting run, this will show the claim. We can restrict our attention to the values of k that are less than or equal to n as we have $f(i) \geqslant i$ for every $i \in \mathbb{N}$. There can be two cases why we do not have $f(k) = n$ for some $k \leqslant n$. Either, w^k is not a bad prefix of ψ, i.e., $f(k)$ is undefined. In such a case, as there is no accepting run of w^k in $\mathcal{A}''$ as $\mathcal{A}''$ accepts only bad prefixes, this means that we cannot loop in the initial state of $\mathcal{A}''$ k times and still obtain a rejecting run of $\mathcal{A}'''$ as it would need to be the same run as an accepting run in $\mathcal{A}''$, which do not exist for w^k. The other possibility for not having $f(k) = n$ is that we have $f(k) = n'$ for some $n' < n$. In such as case, as $\mathcal{A}''$ does not accept words that have strict prefixes that are bad prefixes for ψ, we do not have an accepting run of $\mathcal{A}''$ for w^k. Again, this means that we do not have a rejecting run of $\mathcal{A}'''$ for $w_0 \dots w_n$ that loops in the initial state k times. Since we have excluded all possible values of k, it follows that w does not have a rejecting run of length n for $\mathcal{A}'''$. □

Note that in order to use Theorem 17 for obtaining a GlCA for some LTL property $\mathsf{FG}\psi$, we need to have that ψ is a safety property **and** that ψ can be represented as a non-deterministic very-weak automaton. There are a few LTL properties for which this constraint on ψ does not hold, but $\mathsf{FG}\psi$ can nevertheless be translated to a non-deterministic very-weak automaton over infinite words.

One such example is $\mathsf{FG}(x \rightarrow \mathsf{FG}x)$. The sub-formula $x \rightarrow \mathsf{FG}x$ is not a safety property. Yet, the LTL property $\mathsf{FG}(x \rightarrow \mathsf{FG}x)$ is representable as an equivalent non-deterministic very-weak automaton that is depicted in Figure 14.4.

Thus, such properties have to be translated to DCWs separately in a (ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis approach. We translate the remaining properties to GlCAs using the construction of Theorem 17. The following lemma shows that GlCAs are closed under conjunction without blowup, which allows us to first take the conjunction of the GlCAs for the co-Büchi type properties in the guarantees or assumptions of a specification, and then translate the result to a single DCW using the construction from Lemma 18.

Lemma 20. *Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ and $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', F')$ be a globalised co-Büchi word automata. We can compute a GlCA $\mathcal{A}'' = (Q'', \Sigma, \delta'', Q_0'', F'')$ that accepts all words that are accepted by both $\mathcal{A}$ and $\mathcal{A}'$ by taking:*

- $Q'' = Q \cup Q'$ *(w.l.o.g, we assume that Q and Q' are disjoint),*
- *For all $q \in Q$ and $x \in \Sigma$, we set $\delta''(q,x) = \delta(q,x)$. For all $q \in Q'$ and $x \in \Sigma$, we have $\delta''(q,x) = \delta'(q,x)$,*
- $Q''_0 = Q_0 \cup Q'_0$, *and*
- $F'' = F \cup F'$.

Proof. The definition ensures that the set of rejecting runs of $\mathcal{A}''$ is the union of the rejecting runs of $\mathcal{A}$ and $\mathcal{A}'$ (for every word). Thus, if either $\mathcal{A}$ or $\mathcal{A}'$ reject a word, there are infinitely many rejecting runs, and $\mathcal{A}''$ has these rejecting runs as well. On the other hand, if $\mathcal{A}''$ has infinitely many rejecting runs for a word, then either $\mathcal{A}$ or $\mathcal{A}'$ have an infinite number of rejecting runs for the word. □

To sum up how all the automaton constructions fit together in (ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis, Figure 14.5 describes how the assumptions or guarantees are processed before building a synthesis game according to Definition 59.

14.3 Making use of the two players – choosing the next obligations

To conclude this chapter, let us focus on the similarities of GRabin(1) and ACTL ∩ LTL synthesis and discuss an optimisation concept for the games built in both of these synthesis approaches as well as their combination, which we described in Chapter 14.2.

In all these synthesis approaches, counters are introduced into the games for cycling through the liveness obligations (components i and j in Definition 46, components i and j in Definition 57, and the counters introduced by the construction of Proposition 6 when translating the UVW to a DBW as needed for building the synthesis game in Definition 59). These proceed one-by-one, and whenever the counter has cycled through all of the indices, a position with colour 1 or 2 is visited.

The mapping from obligation states to counter values that represent waiting for the state to be left is rather arbitrary. This fact can delay solving a synthesis game. If it is only the last liveness obligation that a system cannot fulfil, then the fact that for the last counter value, we cannot make progress, has to be propagated through all the other counter values in the game solving process. We can however do better than this.

The main idea is to let the two players choose the next obligation for the respective other player. So whenever normally the liveness guarantee counter would be increased, the environment player can choose the next obligation for the system. On the other hand, if a liveness assumption obligation has just been fulfilled, the system player should choose which of these obligations to fulfil next.

Henzinger and Piterman (2006) applied a similar idea to simplify the determinisation of non-deterministic Büchi automata: they define *good for games* automata, which are non-deterministic parity automata that allow spreading them to a synthesis game that is winning if and only if the specification represented by the automaton is realisable. Thus, these parity automata are, in a sense, well-enough determinised to allow the system player to resolve the non-determinism in the parity automaton eagerly without changing the realisability of the overall specification. Their approach and the one here are similar in the way that we assign an additional task to the system player that she is guaranteed to be able to perform in a synthesis game for realisable specifications.

The edge functions of the two players in a game and its colouring function become a bit more complicated with our modification. Let $\mathcal{A} = (V^0, V^1, E^0, E^1, \Sigma^0, \Sigma^1, v_{init}, c)$ be a parity game that is built according to Definition 46 (for generalised Rabin(1) synthesis), Definition 57 (for ACTL ∩ LTL synthesis), or Definition 59 (for their combination). For an edge in E^1, in which basically all counters, flags, and automaton state updates are performed, we need to allow changing the i counter value (for the assumptions) to an arbitrary value if the position witnesses the satisfaction of the respective assumption obligation, and give that position a colour of 1 (if not higher). The system player can choose the next assumption obligation. Choosing new guarantee obligations on the other hand has to be performed by the environment player, and we must again enforce that the environment player can only change the next obligation if the current obligation has just been fulfilled, as otherwise the environment player could prevent visiting a position with colour 2 at all by switching back and forth between obligations that need some kind of preparation phase by the system player to be fulfilled.

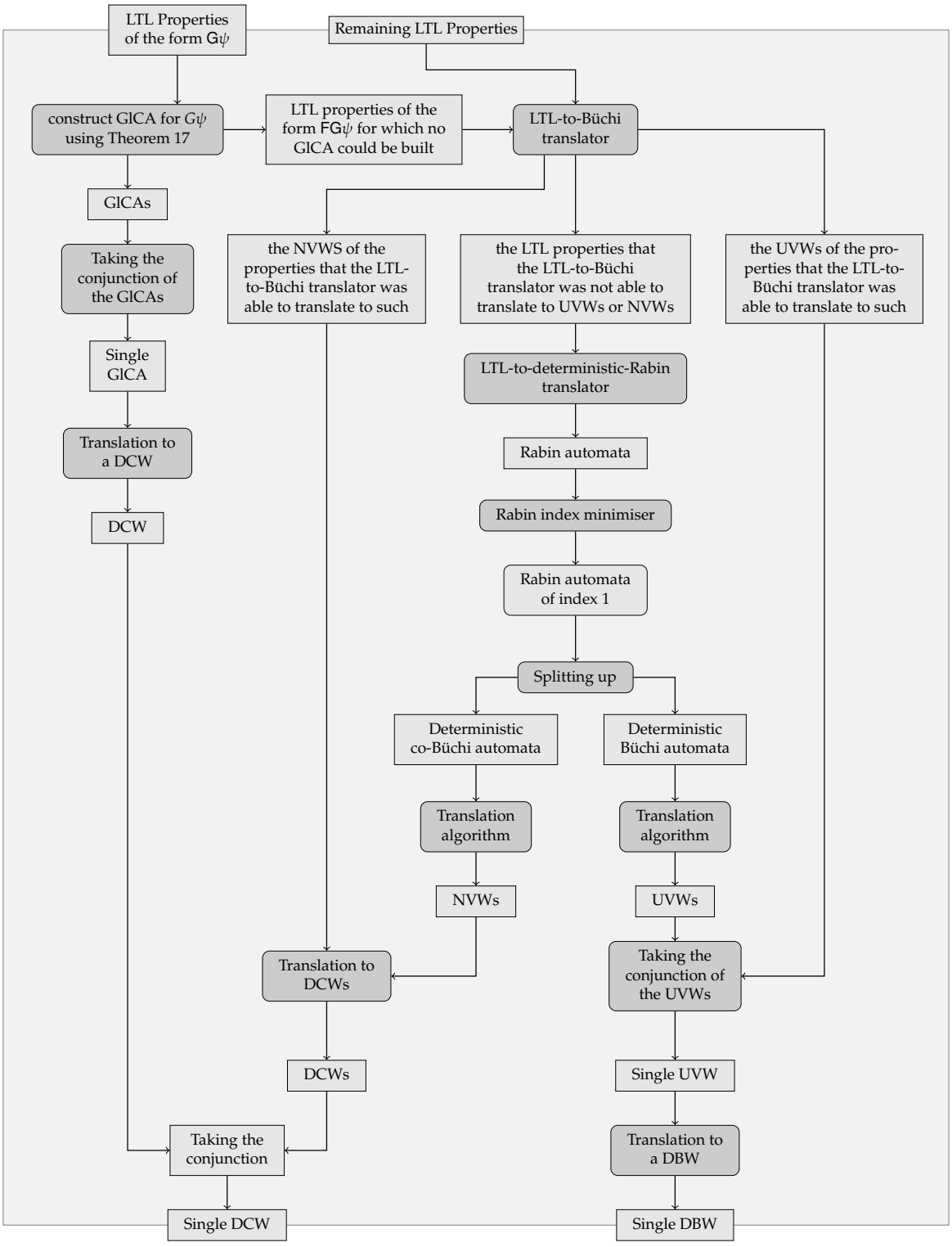

Figure 14.5: Overall workflow for processing the assumptions or guarantees in (ACTL ∩ LTL) ∪ co-(ACTL ∩ LTL) synthesis for robust synthesis, including the ruggedisation step

Note that in the resulting game, there are plays that are winning for the system player, but do not correspond to words that are models of the specification. The reason is that the environment player is not forced to iterate infinitely often over every possible pointer value. Thus, a winning strategy for the system player in this game does not correspond one-to-one to a Mealy machine that satisfies the specification. For obtaining an implementation for the specification, we need to apply some post-processing for a system player's winning strategy in the parity game.

The post-processing step is however not difficult: observe that the worst case for the system player is that the environment player cycles through all of the guarantee pointer values. This way, the system player can only win a play if the decision sequences for the play represent a model of the guarantees or if the system player is able to eventually point out some liveness assumption that does not hold. In both cases, the specification is met. Thus, if we attach a round-robin counter for the assumption pointer to a system player's strategy, we obtain a valid result for our synthesis problem.

A concrete definition of an ACTL ∩ LTL synthesis game using this construction has been given in a paper by Ehlers (2012a).

Chapter 15

Conclusion

In this part of this thesis, we dealt with the problem of identifying practically relevant specification languages from which we can efficiently perform synthesis, and provided suitable solutions to the problem.

We started with generalised Rabin(1) synthesis, an extension of generalised reactivity(1) synthesis and showed that the concept maximises the specification fragment under which we can still use constant-colour parity games as computation model, which in turn allows a fixed-point-based solution algorithm that can easily be applied in a symbolic manner. When compared to generalised reactivity(1) synthesis, the extended specification fragment allows us to use the approach for the synthesis of *robust* systems that behave reasonably under assumption violations.

Then, we discussed ACTL $\cap$ LTL synthesis, which extends generalised reactivity(1) synthesis by not only making pre-synthesis unnecessary, but also helps with retaining the structure of a problem in the synthesis parity game.

Finally, we discussed the combination of the two approaches, and have seen that by applying some minor modifications, we obtain a synthesis approach that produces parity games with a nice structure that have winning strategies which represent robust implementations of the underlying specification.

The focus in this part of the thesis was on the practical side of synthesis, and driven by the question how we can make synthesis technology more useful in the field. Nevertheless, the results of this part have many implications for the theory of the field: we have established a tight bound on how far the generalised reactivity(1) idea can be pushed, and obtained important results on universal very-weak automata, which unify the simplicity of reasoning over them in synthesis with only linear blow-up under conjunction.

As a side-results, we also obtained a procedure to translate an LTL property to an ACTL property, whenever possible. For the converse direction, i.e., from (A)CTL to LTL, how to perform such a translation was well-known since the identification of the translatable cases by Clarke and Draghicescu (1988). Nowadays, their result is taught in many formal methods lecture series. Previously, the result was typically accompanied by the addition that no algorithm for the converse direction is known. With the new results from Chapter 13.2, these times are finally over.

In this thesis, we focused on methods for fast synthesis from subsets of LTL. Many works on applications for reactive synthesis (e.g., Kress-Gazit et al., 2007, 2009; Wongpiromsarn et al., 2010a; Ozay et al., 2011; Wongpiromsarn et al., 2010b; Bloem et al., 2007b,a; Xu et al., 2012) imply that full LTL is typically not needed in practice, but the better scalability of methods for subsets of LTL is useful. However, there is no fundamental difference between the case of full LTL and the case of allowing only a subset of LTL. Thus, many results from this thesis can be used for synthesis from full LTL as well.

For example, Ehlers (2010a) describe an approach to full LTL synthesis in which safety- and non-safety parts of assumptions and guarantees are handled separately. With the results on ACTL $\cap$ LTL synthesis, the idea can be extended to all properties that are representable as UVWs, and thus leaves very few of the properties in a specification to be handled by a the part of the approach that is concerned with the more complicated properties.

Likewise, ideas from full LTL synthesis can be used for synthesis approaches that are not concerned with full LTL. For the techniques in this thesis, of particular interest is the usage of *anti-chains* that has been introduced by Filiot et al. (2009) for full LTL synthesis. In their approach, sets of anti-chains represent pre-fixed points when solving a synthesis game. For full LTL, they needed to augment the chains with counters. Interestingly, the idea can be applied to ACTL $\cap$ LTL synthesis or (ACTL $\cap$ LTL) $\cup$ co-(ACTL $\cap$ LTL) synthesis without the counters, which is a promising approach to improve synthesis performance even more in the future.

Part IV

Experiments & closure

Chapter 16

Efficient symmetric synthesis

In this chapter, we complement the concepts and algorithms introduced in Part II and Part III of this thesis by an experimental evaluation of their applicability. For conciseness, we focus on the combination of the concepts, i.e., efficient symmetric synthesis. The discussion in the following also serves as an example of how symmetric synthesis can assist in the analysis of applications that have an inherent symmetry, and how a specification for a scenario evolves in a specification engineering process that is supported by synthesis technology. Instead of building a single tool-chain, we focus on showing how implementations of individual techniques can also be used with the other concepts of this thesis, thus demonstrating the flexibility of the approaches. We consider two applications for synthesis:

1. the traffic light controller as illustrated in Figure 8.8 on page 89, and
2. the rotation sorter as illustrated in Figure 8.9 on page 90.

We start by describing the settings in Chapter 16.1, and then use these in Chapter 16.2 as specifications to our symmetric synthesis procedure.

16.1 Scenarios for symmetric synthesis

16.1.1 Traffic light system

We model a distributed, rotation-symmetric controller for the traffic lights at a standard 4-way junction. For every direction, there is a sensor for detecting waiting cars, and there are red and green lights in the traffic light boxes. The processes that compute which red and green lights should be switched on are arranged along a simple cycle of length four, just like in Figure 9.1 on page 97. In contrast to that figure, we however have two output signals per process and not only one.

Formally, we define $\mathsf{AP}_L^I = \{carSensed\}$ and $\mathsf{AP}_L^O = \{red, green\}$. The global output alphabet of the overall system that we want to synthesise is thus $\mathcal{I} = 2^{\{carSensed_0, carSensed_1, carSensed_2, carSensed_3\}}$ and $O = 2^{\{red_0, red_1, red_2, red_3, green_0, green_1, green_2, green_3\}}$.

The specification for this setting is the following:

$$
\begin{aligned}
\psi &= \psi_X \wedge \psi_S \wedge \psi_L \text{ for} \\
\psi_X &= \mathsf{G}(red_0 \leftrightarrow \neg green_0) \wedge \mathsf{G}(red_1 \leftrightarrow \neg green_1) \wedge \mathsf{G}(red_2 \leftrightarrow \neg green_2) \\
&\quad \wedge \mathsf{G}(red_3 \leftrightarrow \neg green_3), \\
\psi_S &= \mathsf{G}(green_0 \rightarrow (\neg green_1 \wedge \neg green_3)) \\
&\quad \wedge \mathsf{G}(green_1 \rightarrow (\neg green_2 \wedge \neg green_0)) \\
&\quad \wedge \mathsf{G}(green_2 \rightarrow (\neg green_3 \wedge \neg green_1)) \\
&\quad \wedge \mathsf{G}(green_3 \rightarrow (\neg green_0 \wedge \neg green_2)), \text{and} \\
\psi_L &= \mathsf{G}(carSensed_0 \rightarrow \mathsf{F} green_0) \wedge \mathsf{G}(carSensed_1 \rightarrow \mathsf{F} green_1) \\
&\quad \wedge \mathsf{G}(carSensed_2 \rightarrow \mathsf{F} green_2) \wedge \mathsf{G}(carSensed_3 \rightarrow \mathsf{F} green_3)
\end{aligned}
$$

This specification consists of three parts: ψ_X ensures that the red and green lights are mutually exclusive for every direction, and for every traffic light box, there is always either the red light or the green light switched on. The specification part ψ_S implements the usual safety constraint that we shall never have a green light for one direction and its neighbouring direction at the same time. Finally, ψ_L describes the liveness constraint that cars that have been sensed by the system should eventually get a green light.

16.1.2 Rotation sorter

As the second scenario, we model a rotation sorter with three incoming and outgoing belts, as depicted in Figure 8.9 on page 90. For every incoming conveyor belt, a barcode reader detects whether a packet shall be forwarded to the next outgoing belt, the next next outgoing belt, or be carried by the rotating platform by one full rotation and pushed to the outgoing conveyor belt next to the to one that delivered it.

The in-belts are controlled, so that the sorter can decide when to drop a packet onto the platform. There are pushers in front of every outgoing belt to push the packet from the platform.

The processes of the symmetric architecture form a simple cycle of length three. We have $\mathsf{AP}_L^I = \{a, b\}$, where a and b together encode whether no packet is waiting at some in-belt ($a = \textbf{false}, b = \textbf{false}$), a packet is waiting to be forwarded by 120 degrees before delivery ($a = \textbf{false}, b = \textbf{true}$), a packet is to be forwarded by 240 degrees before delivery ($a = \textbf{true}, b = \textbf{false}$), or it is to be forwarded by 360 degrees before delivery ($a = \textbf{true}, b = \textbf{true}$). As local output of the processes, we have $\mathsf{AP}_L^O = \{belt, pusher\}$. The former proposition is used to ask the in-belt to deliver the next packet to the platform, while the latter proposition activates the pusher.

The specification essentially states that packets must always eventually be forwarded correctly when they have been sensed. We can express this requirement in LTL by stating:

$$
\begin{aligned}
\psi &= \psi_F, \text{ where} \\
\psi_F &= \mathsf{G}((\neg a_0 \wedge b_0) \rightarrow (\neg belt_0 \,\mathsf{U}\, (belt_0 \wedge \mathsf{XX} pusher_1))) \\
&\quad \wedge \mathsf{G}((a_0 \wedge \neg b_0) \rightarrow (\neg belt_0 \,\mathsf{U}\, (belt_0 \wedge \neg\mathsf{XX}\, pusher_1 \wedge \mathsf{XXXX}\, pusher_2))) \\
&\quad \wedge \mathsf{G}((a_0 \wedge b_0) \rightarrow (\neg belt_0 \,\mathsf{U}\, (belt_0 \wedge \neg\mathsf{XX}\, pusher_1 \wedge \neg\mathsf{XXXX}\, pusher_2 \\
&\qquad \wedge \mathsf{XXXXXX}\, pusher_0)))
\end{aligned}
$$

By the symmetry of our setting, we do not also have to declare that packets from the incoming belts 1 and 2 are forwarded correctly, as in the first step of the symmetric synthesis process, the specification is replicated for all rotations anyway.

16.2 Experimental results

We implemented a couple of tools that jointly allow us to evaluate the the applicability of the concepts and algorithms of this thesis. In particular, these are:

- A translation workflow from linear-time temporal logic (LTL) to universal very-weak automata (UVW), consisting of:
  - a translator from deterministic Rabin automata to deterministic Büchi automata, whenever possible,
  - a SAT-based minimiser for deterministic Büchi automata (see Chapter 12.2.1),
  - a translator from a deterministic Büchi automaton to a universal very weak automaton, implementing the algorithm from Chapter 13, and
  - an orchestrator that uses these three tools to manage the workflow. If first calls the tool `ltl3ba` (Babiak et al., 2012) to check if this LTL-to-Büchi translator is already able to produce a UVW for a given LTL formula. If not, the tool `ltl2dstar` (Klein and Baier, 2006) is used to obtain a deterministic Rabin automaton for the LTL formula. Afterwards, it is translated to a deterministic Büchi automaton using the procedure by Krishnan et al. (1995). If this is not possible, then we know that there exists no equivalent UVW and abort. Otherwise, we minimise the automaton, and finally translate it to a UVW.
- A binary decision diagram-based game solver for synthesis games built according to Definition 57, including the idea from Chapter 14.3
- A GRabin(1) game solver based on binary decision diagrams

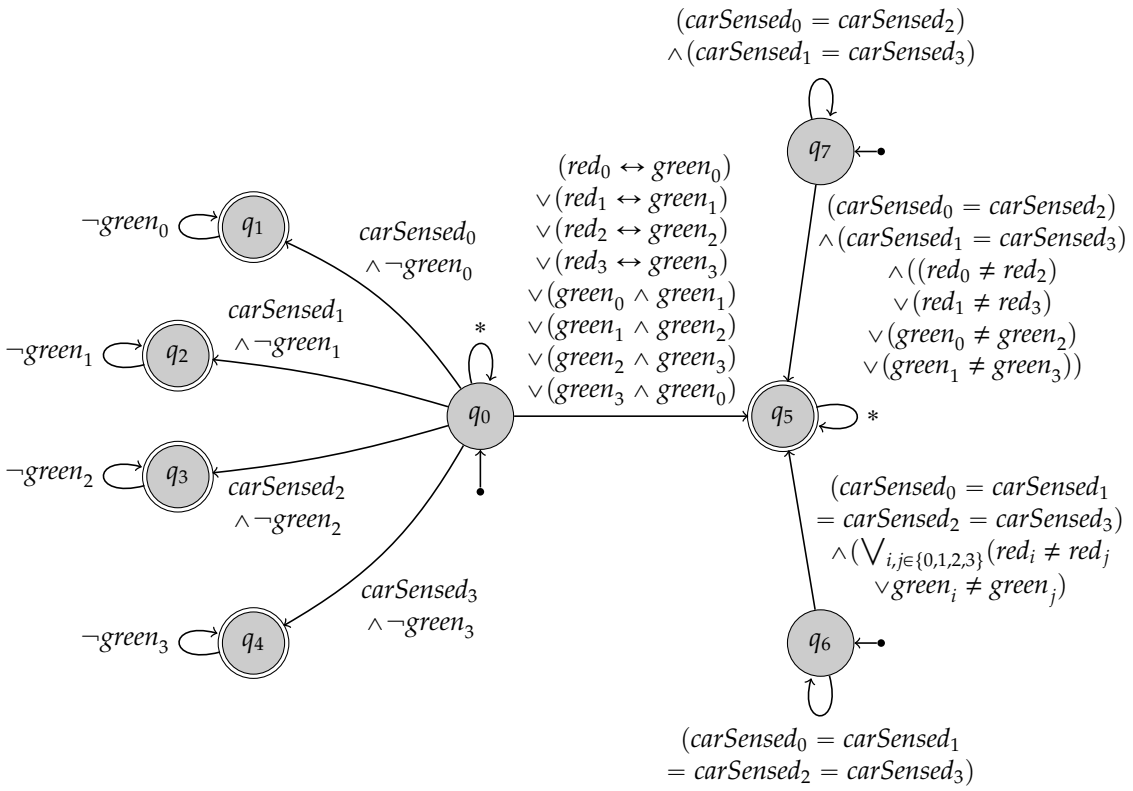

Figure 16.1: A universal very-weak automaton for the basic specification of the traffic light system, including the automaton part needed for checking the absence of introduction of asymmetry by the system.

The first two items on this list together form a complete workflow for ACTL $\cap$ LTL synthesis.

All experimental results in the following have been obtained on a AMD E-450 1.6GHz dual-core computer with 4GB RAM running a x86-version of Linux. All computation steps are single-threaded.

16.2.1 Traffic light

In the symmetric synthesis algorithm from Chapter 9.1.4, the first step is to compute the modified specification $\psi' = \psi \wedge \mathrm{rot}(\psi, 1) \wedge \ldots \mathrm{rot}(\psi, n-1)$ for ψ being the original specification and n being the number of processes. In the basic traffic light scenario, the specification ψ' is actually equivalent to ψ, so this step is saved here.

For realisability checking of the setting, we can thus use ψ directly and feed it together with the automaton that checks for the absence of asymmetry introduction to the ACTL $\cap$ LTL workflow. For simplicity, we encoded the acceptance by the latter automaton in LTL.

The workflow only needs 1.1 seconds to conclude the unrealisability of the setting. The UVW built for the overall specification along the way is depicted in Figure 16.1.

The unrealisability of the combination of architecture and specification can be explained by the fact that all car sensors may be triggered at the same time, and then there is not enough asymmetry in the input to allow setting some traffic light to green and another traffic light to red at the same time.

To fix this problem, we consider the case that we add an assumption that requires that at some point, not all sensors produce the same values. The modified specification is then of the form

$$\psi'' = \psi_B \rightarrow (\psi_X \wedge \psi_S \wedge \psi_L),$$

where we define:

$$\begin{aligned} \psi_B \quad = \quad & \mathsf{F}((carSensed_0 \vee carSensed_1 \vee carSensed_2 \vee carSensed_3) \\ & \leftrightarrow \neg(carSensed_0 \wedge carSensed_1 \wedge carSensed_2 \wedge carSensed_3)) \end{aligned}$$

In the first step of the symmetric synthesis workflow, this specification is now replicated for all rotations. The result is a conjunction of implications, which is not immediately applicable in the ACTL $\cap$ LTL synthesis workflow. As all rotations are equivalent, we can remove them however, resulting in a specification that is suitable for ACTL $\cap$ LTL synthesis. The requirement that the system must not introduce asymmetry into its run can also be incorporated as a simple guarantee in the synthesis process, as the assumption is a pure liveness property and thus, the system under design cannot benefit from its non-satisfaction in finite time.

The workflow only needs 1.2 seconds to conclude the realisability of the modified specification and to compute an implementation (for all processes together). Note that the added assumption does not guarantee that eventually, the symmetry is broken completely by the input. For example, the assumption is already satisfies if at some point, the overall system to be synthesised obtains the input $\{carSensed_0 \mapsto \textbf{false}, carSensed_1 \mapsto \textbf{true}, carSensed_2 \mapsto \textbf{false}, carSensed_3 \mapsto \textbf{true}\}$. This way, the symmetry is only broken partially, but this suffices for this scenario, as we can always trigger $green_0$ and $green_2$ at the same time, just as $green_1$ and $green_3$ can always be set to **true** at the same time.

Requiring only one light to be green at a time would change this fact. We can capture this case by the specification

$$\psi''' = \psi_B \rightarrow (\psi_X \wedge \psi_S \wedge \psi_L \wedge \psi_O),$$

where we define:

$$\psi_O \quad = \quad \mathsf{G}(\neg green_0 \vee \neg green_2) \wedge \mathsf{G}(\neg green_1 \vee \neg green_3)$$

The unrealisability of ψ'' in the rotation-symmetric architecture of our traffic light system is proven by the ACTL $\cap$ LTL synthesis workflow in 1.25 seconds.

16.2.2 Rotation sorter

The basic scenario of the rotation sorter contains no assumptions, so we can immediately feed the specification $\psi = \psi_F$ as defined in Chapter 16.1.2 to the ACTL $\cap$ LTL workflow after adding the rotated versions of ψ_F to ψ and adding the requirement that the system shall not introduce asymmetry.

Feeding the scenario to our ACTL $\cap$ LTL synthesis workflow, it computes the UVW depicted in Figure 16.2. The scenario is found to be unrealisable after 130 seconds. A closer analysis reveals that this is already true without the requirement that the solution shall not introduce asymmetry (24.1 seconds of computation time are needed for checking this case).

The reason for unrealisability is the fact that we did not require that the environment cannot change the destination of a packet while the packet is waiting for the belt to be triggered. By switching the destination in such a case, the environment of the system can enforce that two conjuncts in ψ_F are triggered for the same packet, and the right-hand side of the implications that a conjunct represents are in conflict. On the other hand, the system can also not always trigger a belt immediately, as a packet to a different designation might be passing by on the platform at that point – pushing the packet onto the belt would then lead to at least one packet being delivered incorrectly, as they cannot be separated once they are at the same place on the platform.

The solution to this problem is to add the assumption on the environment that the destination of a packet is never changed, i.e., while a packet is waiting, the values of a and b do not change.

The modified specification is $\psi' = \psi_A \rightarrow \psi_F$, where we define:

$$\begin{aligned}\psi_A \quad = \quad & \mathsf{G}((\neg a_0 \wedge b_0 \wedge \neg belt0) \rightarrow \mathsf{X}(\neg a_0 \wedge b_0)) \\ & \wedge \mathsf{G}((a_0 \wedge \neg b_0 \wedge \neg belt0) \rightarrow \mathsf{X}(a_0 \wedge \neg b_0)) \\ & \wedge \mathsf{G}((a_0 \wedge b_0 \wedge \neg belt0) \rightarrow \mathsf{X}(a_0 \wedge b_0))\end{aligned}$$

In the first step of the symmetric synthesis process, this new specification $\psi' = \psi_A \rightarrow \psi_F$ is processed to:

$$\hat{\psi}' = (\psi_A \rightarrow \psi_F) \wedge \mathrm{rot}(\psi_A \rightarrow \psi_F, 1) \wedge \mathrm{rot}(\psi_A \rightarrow \psi_F, 2)$$

Note that $\hat{\psi}'$ does not fall into the specification fragment supported by ACTL $\cap$ LTL synthesis, as the overall specification is not an implication between a conjunction of assumptions and a conjunction of guarantees any more. We can however weaken $\hat{\psi}'$ as follows:

$$\tilde{\psi}' = (\psi_A \wedge \mathrm{rot}(\psi_A, 1) \wedge \mathrm{rot}(\psi_A, 2)) \rightarrow (\psi_F \wedge \mathrm{rot}(\psi_F, 1) \wedge \mathrm{rot}(\psi_F, 2))$$

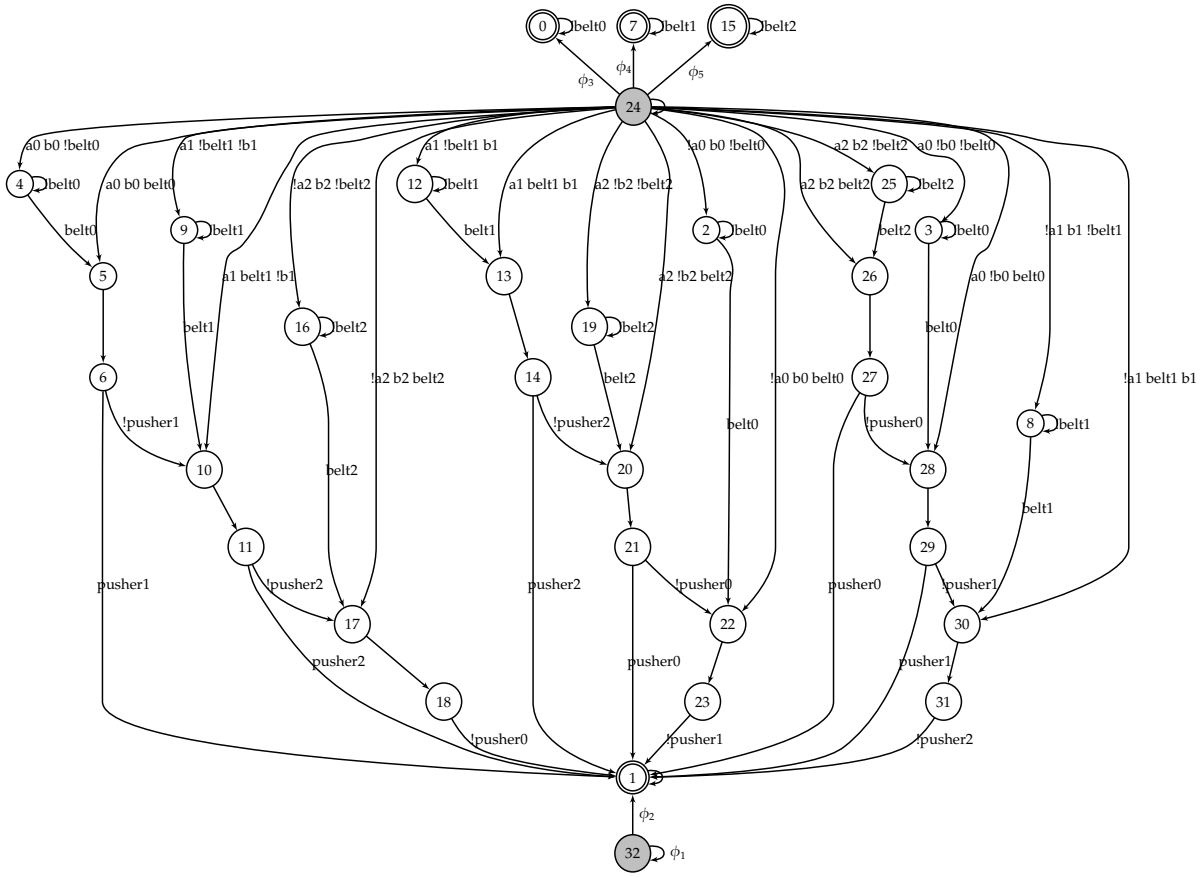

Figure 16.2: A universal very-weak automaton for the basic specification of the rotation sorter system. Initial states are marked in grey colour. Rejecting states are doubly-circled. Some edge labels have been abbreviated for space reasons. In particular, we have $\phi_1 = (a_0 = a_1 = a_2) \wedge (b_0 = b_1 = b_2)$, $\phi_2 = \phi_1 \wedge ((pusher_0 \neq pusher_1) \vee (pusher_1 \neq pusher_2) \vee (pusher_0 \neq pusher_2) \vee (belt_0 \neq belt_1) \vee (belt_1 \neq belt_2) \vee (belt_0 \neq belt_2))$, $\phi_3 = \neg belt_0 \wedge (a_0 \vee b_0)$, $\phi_4 = \neg belt_1 \wedge (a_1 \vee b_1)$, and $\phi_5 = \neg belt_2 \wedge (a_2 \vee b_2)$.

Thus, instead of requiring for every belt and pusher individually that packets are forwarded correctly unless at some point the sensor changes its output in an illegal way, we allow all of the processes to stop proper operation once any of the sensors spontaneously changes its value. This reinterpretation is justified by the role of the assumptions – they specify constraints on the behaviour of the environment that we can take for granted. If we allow a part of the system to behave in unexpected ways in case of assumption violations, then we can just as well allow the whole system to behave in this way. We can also just add the requirement that the system shall not introduce asymmetry as a guarantee to the resulting specification as we then require that the implementation cannot introduce asymmetry until the assumptions are violated. So the implementation for one process that we obtain in this setting will still work correctly when instantiated in the symmetric architecture, and we can apply the ACTL $\cap$ LTL synthesis workflow again.

The modified specification is found to be realisable after 16 minutes and 18 seconds (including the time to compute an implementation).

As in this scenario, we have assumptions over the environment that can be temporarily violated, we can use the scenario in a GRabin(1) synthesis tool in order to synthesise a system that is robust with respect to assumption violations. We first decompose the guarantees into safety and liveness aspects, and obtain:

$$\psi_F'' \quad = \quad \mathsf{G}((a_0 \vee b_0) \rightarrow \mathsf{F}\, belt0)$$

$$\wedge\, \mathsf{G}((\neg a_0 \wedge b_0) \rightarrow (\neg belt_0\, \mathsf{W}\, (belt_0 \wedge \mathsf{X}\mathsf{X} pusher_1)))$$
$$\wedge\, \mathsf{G}((a_0 \wedge \neg b_0) \rightarrow (\neg belt_0\, \mathsf{W}\, (belt_0 \wedge \neg\mathsf{X}\mathsf{X}\, pusher_1 \wedge \mathsf{X}\mathsf{X}\mathsf{X}\mathsf{X}\, pusher_2)))$$
$$\wedge\, \mathsf{G}((a_0 \wedge b_0) \rightarrow (\neg belt_0\, \mathsf{W}\, (belt_0 \wedge \neg\mathsf{X}\mathsf{X}\, pusher_1 \wedge \neg\mathsf{X}\mathsf{X}\mathsf{X}\mathsf{X}\, pusher_2$$
$$\wedge\mathsf{X}\mathsf{X}\mathsf{X}\mathsf{X}\mathsf{X}\mathsf{X}\, pusher_0)))$$

In these equations, the *weak until operator* that is often defined for LTL has been used. Formally, given some LTL sub-formulas ϕ and ϕ', we say that $\phi\, \mathsf{W}\, \phi'$ holds whenever $\mathsf{G}\phi$ or $\phi\, \mathsf{U}\, \phi'$ hold. Note that the only conjunct of ψ''_F that is now still a liveness property is $\mathsf{G}((a_0 \vee b_0) \rightarrow \mathsf{F}\, belt0)$.

Afterwards, we are ready to ruggedise the resulting specification

$$\tilde{\psi}'' = (\psi_A \wedge \mathrm{rot}(\psi_A, 1) \wedge \mathrm{rot}(\psi_A, 2)) \rightarrow (\psi''_F \wedge \mathrm{rot}(\psi''_F, 1) \wedge \mathrm{rot}(\psi''_F, 2) \wedge \psi_N),$$

where ψ_N is an LTL formula that encodes the acceptance of the system's behaviour by the automaton that checks for the absence of asymmetry introduction by the system.

We then ruggedise $\tilde{\psi}''$ as described in Chapter 12.4 and use the concepts of Chapter 14.2 to efficiently get from the resulting specification to automata. The final encoding into the simpler specification form defined in Chapter 12.1, as needed for our implementation of the approach, has been performed by hand.

The game that our GRabin(1) synthesis implementation builds from the final form of the specification actually only has four colours, as there are no liveness assumptions in this specification, and thus, there are no positions with colour 0 in the game. Including writing the binary decision diagrams (BDDs) that represent a (non-deterministic) strategy for winning the game, which can be used as input to a strategy execution program that simulates a correct controller for the processes in the setting, the computation takes 759.9 seconds. Note that the ACTL $\cap$ LTL synthesis tool uses a more elaborate strategy extraction approach (i.e., the method by Kukula and Shiple, 2000), which explains the fact that the GRabin(1) tool is actually faster on this scenario.

Chapter 17

Conclusion

This thesis presents progress on making the vision of useful and efficient reactive synthesis from industrial-sized specifications come true.

We started with a thorough analysis of the problem of *symmetric synthesis*, where we search for a single implementation for all processes in a distributed architecture such that the resulting overall system satisfies a given specification. Such systems have the favourable property of having more *structure* than monolithic implementations, which improves their understandability and simplifies their certification. We identified the driving factors for the decidability and undecidability of symmetric architectures. We showed that feeding one process' output to another process' input should be avoided in order to keep an architecture decidable, as even the *S0 architecture*, which is the simplest architecture in which this is the case, has an undecidable symmetric synthesis problem. The key to proving this was the definition of a word compression function that allows us to squeeze the values of many signalling bits into one, and reducing the symmetric realisability problem for the more complicated *S2 architecture* to the simpler one. For showing the undecidability of symmetric synthesis for the S2 architecture, we constructed a scenario in which the processes do not obtain enough information to guess their identity correctly, and reduce an undecidable non-symmetric distributed synthesis problem to our case.

On the positive side, we identified a large class of symmetric architectures with a decidable synthesis problem, and provided complexity-theoretically optimal synthesis procedures for both linear-time and branching-time specifications. The key idea employed in the construction is that instead of synthesising the implementation for one process, we can synthesise an implementation for all processes running in parallel and impose the requirement that the solution shall have the *symmetry property*. This property is non-regular, and we showed how to decompose it into a regular one that we can incorporate into the specification of the system under design, and a non-regular one that can be dealt with on the automaton level. We conjecture that this novel decomposition scheme has applications far beyond symmetric synthesis.

After dealing with symmetric synthesis, we turned towards the question of how to perform scalable reactive synthesis in practice. We started by discussing the properties of the efficient, but relatively limited, generalised reactivity(1) synthesis approach that has been used in a plethora of practical applications. We identified the reasons for this success and the two shortcomings that this approach still has: *limited expressivity* and the *need for pre-synthesis*.

To tackle the first of these problems, we defined a new synthesis approach, named *generalised Rabin(1) synthesis*, that can accommodate all specifications consisting of assumptions and guarantees that can all be represented by deterministic Rabin automata with one acceptance pair. These in particular include the class of *stability properties*, which is the main class of properties that are useful in practice and are unsupported in generalised reactivity(1) synthesis. We showed that our new synthesis approach still inherits the nice properties of generalised reactivity(1) synthesis, and that any further significant extension would lead to losing these properties. The expressivity extension by stability properties allows us to apply the approach to the synthesis of *robust systems*, which operate reasonably even if the assumptions about the environment that the system to be synthesised has to work in are violated.

To counter the problem of pre-synthesis, we proposed a synthesis workflow for specifications whose assumptions and guarantees are representable both in universally path-quantified branching-time logic (ACTL) and linear-time temporal logic (LTL). The key element in the approach is a novel construction to translate from LTL to *universal very-weak automata* (UVW), which is the characterising automaton class for this intersection between ACTL and LTL. Thus, we allow the user to use full LTL for writing the specification, and only require that the sub-properties of the specification are representable somehow in ACTL as well. This allows us to build a synthesis approach around universal very-weak automata, which have a synthesis problem that is only exponential in the size of the automaton. At the same time, we provide the user with a high-level language to write the specification in. As a corollary of the

new LTL-to-UVW translation construction, we also obtained the first procedure to translate from LTL to ACTL, whenever possible. While the other translation direction is taught in many courses on formal methods nowadays, a translation method from LTL to ACTL (whenever possible) that we obtained here was previously unknown.

We tied together the results on symmetric and efficient synthesis by using the techniques developed in this thesis to analyse two application scenarios: a traffic light system and a packet sorter controller. Our experiments show the applicability of the new techniques and concepts and describe how symmetric synthesis technology can assist a system designer with analysing an application scenario under concern.

The results of this thesis are complemented by a self-contained and easily-accessible introduction to the field of reactive synthesis, which can also be read in isolation. Its aim is to serve as a reference for researchers from other fields that want to dive into reactive synthesis.

17.1 Outlook

This thesis both opens and closes chapters in the research on the synthesis problem for reactive systems.

In our treatment of the symmetric synthesis problem, we introduce the idea to decompose a non-regular constraint on the systems that we want to synthesise into a regular and non-regular one that we can incorporate into the synthesis workflow by modifying the specification automaton. While symmetry is a favourable property of distributed reactive systems, the same approach could be used to incorporate requirements into the synthesis process for non-distributed systems.

In particular, how to synthesise *understandable implementations* is still largely an open problem. In many contexts in which the correct functioning of a controller is safety-critical, reactive synthesis would be a suitable technique to ensure correctness by construction. Yet, controllers that cannot easily be analysed by engineers tend not to be trusted. By identifying constraints over the structure of an implementation that would imply its understandability by engineers, we could make sure that the systems that we obtain are in fact understandable. The decomposition idea from the symmetric synthesis algorithm shows us that in this enterprise, we do not need to be afraid of ideas that would require to impose non-regular constraints over the synthesised solution. As long as we can decompose the requirement, it can be incorporated.

On the other hand, our discussion of the symmetric synthesis problem also closes a chapter of the research on the synthesis problem. We have identified that the problem is undecidable even for simple architectures, but at the same time characterised a large decidable class of architectures. Our synthesis algorithm for the latter class is complexity-theoretically optimal, leaving no room for further significant improvement.

Our generalised Rabin(1) synthesis approach in a way also closes a chapter on reactive synthesis. It shows how much we can extend the expressivity of synthesis approaches based on solving parity games with a constant number of colours without losing the good properties of these approaches. The supported specification fragment is also closed under the process of ruggedising a specification, making the class a natural choice for the synthesis of robust systems.

Finally, the results that form the basis of our ACTL $\cap$ LTL synthesis approach can be seen as opening a new chapter on synthesis. By closing the gap between a specification logic that is suitable for engineers and universal very-weak automata, we open up this automaton class to be used as a specification model for synthesis. They are conceptually simple and lend themselves to many new approaches to efficiently reason about synthesis games built from them. In the experimental evaluation of this thesis, we used a binary decision diagram based implementation. Yet, universal very-weak automata are well-suited to be used with other symbolic reasoning techniques, too. For example, the structure of the synthesis games built from UVW specifications lends itself to the usage of *anti-chains* (Filiot et al., 2009) in the game solving process. Other approaches based on solving quantified Boolean formulas with specifically adapted solvers are equally conceivable. The fact that we know how to build UVWs for large sets of assumptions and guarantees allow us to reduce the actual synthesis problem to a game solving problem that can be formally described very easily, making UVWs an ideal starting point for the exploration of new symbolic reasoning techniques for solving games without the need to care about the details of the logic used for specification. We have seen how a simple problem model to start with can foster the development of powerful reasoning tools in the realm of *SAT solving*, which has found a wide range of practical applications. It is fair to conjecture that UVWs can serve as the simple problem model that allows the same progress on reasoning engines for the scope of synthesis in the future.

Appendix A

Basic definitions and notation

To improve the accessibility of this thesis to those readers who are not so familiar with the commonly used notation and concepts of computer science and the branches of mathematics used in this thesis, we provide some helping definitions here.

Sets and words The set of *natural numbers* is denoted by $\mathbb{N}$ and includes all positive numbers $\geqslant 0$. The set of *integer numbers* is denoted by $\mathbb{Z}$. The set of *Boolean values* is denoted by $\mathbb{B}$, and consists of the elements $\{\textbf{false}, \textbf{true}\}$. Sometimes, to keep the notation short in examples, we also use the set $\{0, 1\}$ instead of $\{\textbf{false}, \textbf{true}\}$. For $\mathbb{Z}$ and $\mathbb{N}$, we denote the multiplication operation by "$\cdot$". Given an integer number $z \in \mathbb{Z}$ and a natural number $n \in \mathbb{N}$, the *modulo operator* (mod) computes the unique number $0 \leqslant n' < n$ such that $z = n' + i \cdot n$ for some $i \in \mathbb{Z}$. For two sets X and Y, the $\times$ operation defines the *cross-product* between X and Y, i.e., $X \times Y = \{(x, y) \mid x \in X, y \in X\}$. Given some element $(x, y) \in X \times Y$, we use the $|$ operator to map (x, y) to its components, i.e., we have $(x, y)|_X = x$ and $(x, y)|_Y = y$.

Given some set X, we denote by X^* the set of all finite sequences of X, i.e., all *words* $w = w_0 w_1 \dots w_n$ for which for all $i \in \{0, \dots, n\}$, we have $w_n \in X$. Likewise, X^ω denotes the set of all infinite sequences of X, i.e., all infinite words $w = w_0 w_1 \dots$ for which for all $i \in \mathbb{N}$, we have $w_i \in X$. We denote the subword of w starting from the jth element for some $j \in \mathbb{N}$ by w^j, i.e., for some finite word $w = w_0 \dots w_n$, we define $w^j = w_j w_{j+1} \dots w_n$, and for some infinite word $w = w_0 w_1 \dots$, we define $w^j = w_j w_{j+1} w_{j+2} \dots$.

Fixed points Let X be a set. We call some function $f : 2^X \to 2^X$ *monotone* if for every $A, B \subseteq X$ with $A \subseteq B$, we have that $f(A) \subseteq f(B)$. Given some monotone function f, we define the *least fixed point* $\mu X.f(X)$ of f by setting $\mu^0 X.f(X) = \varnothing$, $\mu^{i+1} X.f(X) = f(\mu^i X.f(X))$ for every $i \in \mathbb{N}$, and set $\mu X.f(X) = \mu^k X.f(X)$ for the least value of $k \in \mathbb{N}$ such that $\mu^k X.f(X) = \mu^{k+1} X.f(X)$. Similarly, we define the *greatest fixed point* $\nu X.f(X)$ by setting $\nu^0 X.f(X) = \varnothing$, $\nu^{i+1} X.f(X) = f(\nu^i X.f(X))$ for every $i \in \mathbb{N}$, and set $\nu X.f(X) = \nu^k X.f(X)$ for the least value of $k \in \mathbb{N}$ such that $\nu^k X.f(X) = \nu^{k+1} X.f(X)$. To emphasise that the least and greatest fixed points are fully evaluated, we may denote $\mu X.f(X)$ and $\nu X.f(X)$ as $\mu^\infty X.f(X)$ and $\nu^\infty X.f(X)$, respectively.

When dealing with formulas with nested fixed point operators, an additional rule applies. If a bounded fixed point operator μ^i/ν^i is used in the scope of an unbounded fixed point operator in an equation, then the bound on the inner operator is only applied after the fixed point of the outer operator has been reached. For example, $\nu Y.\mu^j X.f(X, Y)$ is evaluated to $\mu^j X.f(X, \nu Y.\mu X.f(X, Y))$ for every fixed $j \in \mathbb{N}$.

Group theory We call $(S, \circ)$ a *group* if S is a set, $\circ : S \times S \to S$ is an operation, and the following conditions hold:

- For all $x, y, z \in S$, we have $(x \circ y) \circ z = x \circ (y \circ z)$.
- There exists some element $e \in S$ such that for all elements $y \in S$, we have $y = y \circ e = e \circ y$. We call e the *neutral element*.
- For every element $x \in S$, there exists some element $x^{-1} \in S$ such that $x \circ x^{-}1 = e$. We call $x^{-}1$ the *inverse element* of x.

We call $(S, \circ)$ a *finite group* if S is a finite set, and call $(S, \circ)$ an *infinite group* otherwise. For some $n \in \mathbb{N}$, we call the group $(\{0, 1, \dots, n-1\}, \circ)$ the *modulo group* if we define $x \circ y = (x + y) \mod n$ for all $x, y \in \{0, 1, \dots, n-1\}$. In such a case, we also denote the inverse elements of some $x \in \{0, 1, \dots, n-1\}$ by $-x$.

Automata on finite words and regular expressions Automata over finite word are defined as tuples $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states of $\mathcal{A}$.

Automata over finite words accept or reject (finite) words in Σ^*. Notationally, we extend δ to a function $\delta^* : 2^Q \times \Sigma^* \to 2^Q$ by setting $\delta^*(S, x) = \{q' \in Q \mid \exists q \in S : q' \in \delta(q, x)\}$ for $x \in \Sigma$ and $S \subseteq Q$, and set $\delta^*(S, wx) = \delta^*(\delta^*(S, w), x)$ for all $w \in \Sigma^*$ with $|w| \geqslant 1$, $x \in \Sigma$, and $S \subseteq Q$. We say that the automaton accepts a word $w = w_0 \dots w_n \in \Sigma^*$ if $\delta^*(Q_0, w) \cap F \neq \varnothing$. The star notation for extending a transition relation is also used in other contexts, such as for Mealy or Moore machines (see Definition 1 on page 26). The set of words accepted by an automaton is also called its language. If the set of states of the automaton is some structured set, then for some state $q \in Q$, we also refer to the *label of a state* when interpreting the concrete value of q.

If Q_0 is a set of size one and for all $q \in Q$ and $x \in \Sigma$, we have $|\delta(q, x)| \leqslant 1$, then we call $\mathcal{A}$ a *deterministic automaton over finite words* (*DFA*). Otherwise it is a *non-deterministic automaton over finite words* (*NFA*). The former of these classes has the nice property to allow complementation in linear time, i.e., given an automaton $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ with language L, we can obtain an automaton for the language $\Sigma^* \backslash L$ by taking $\mathcal{A}' = (Q, \Sigma, \delta, Q_0, Q \backslash F)$.

The set of languages representable by DFAs and NFAs (which are the same) is called the *regular languages*. Another way to specify regular languages is using *regular expressions*. Given some alphabet Σ, every subset $X \subseteq \Sigma$ is a basic regular expression. Given regular expressions ψ and ψ', we can compose them to more complex regular expressions by taking $\psi \cdot \psi'$, $\psi \cup \psi'$, or ψ^*. On the semantic level, a regular expression denotes a subset of Σ^*. For some regular expression $X \subseteq \Sigma$, we define the semantics $[\![X]\!]$ of X to be X itself. For the regular expression operators, we define their semantics as $[\![\psi \cdot \psi']\!] = \{w \in \Sigma^* \mid \exists w_1, w_2 \in \Sigma^* : w = w_1 w_2, w_1 \in [\![\psi_1]\!], w_2 \in [\![\psi_2]\!]\}$, $[\![\psi \cup \psi']\!] = [\![\psi]\!] \cup [\![\psi']\!]$, and $[\![\psi^*]\!] = \{w \in \Sigma^* \mid \exists w_1 \in \Sigma^*, w_2 \in \Sigma^*, \dots, w_n \in \Sigma^* : w = w_1 w_2 \dots w_n, \forall i \in \{1, \dots, n\} : w_i \in [\![\psi]\!]\}$.

Turing machines *Turing machines* are a basic computation model in computer science. Formally, a (non-deterministic) Turing machine is defined as a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, g)$ with the set of states Q, the input alphabet Σ, the tape alphabet $\Gamma \supseteq \Sigma$, the transition function δ, the initial state q_0, and $g : Q \to \{\mathit{accept}, \mathit{normal}, \mathit{reject}\}$ is a function that marks the states as accepting states, rejecting states, or normal states. The Turing machine also has some designated empty tape symbol $\epsilon \in \Gamma$.

A Turing machine operates on a Turing tape all of whose elements are in Γ. When the machine starts operating, it has some input word $w \in \Sigma^*$ written to the tape, and all other tape cells are initialised with ϵ. The *tape head* starts at the first element of the word, and the Turing machine starts in its initial state q_0. At every step of the computation of the machine, it checks the tape content at its current position and its current state, and applies the transition function $\delta : Q \times \Gamma \to (Q \times \Gamma \times \{L, U, R\})^2$ to obtain an element $(Q \times \Gamma \times \{L, U, R\})^2$ from this combination. It then non-deterministically chooses one of the two elements of the type $(Q \times \Gamma \times \{L, U, R\})$ for its further operation in this step. If the two elements to choose from are always the same, we call the Turing machine deterministic. The elements represent (1) the next state after a transition, (2) the tape alphabet symbol that is written to the current position when taking the transition, and (3) an indicator for selecting whether the tape head moves left for the next step (L), whether the tape head position stays unchanged after the current step (U), or whether the tape head moves right for the next step (R).

While the Turing machine is operating, it produces a run of the machine. If the run at some point reaches a state q with $g(q) = \mathit{accept}$, and did not visit some state q with $g(q) = \mathit{reject}$ beforehand, then we say that the run is accepting and that the word is accepted by the Turing machine. The language of a Turing machine is the set of accepted words.

We say that a Turing machine is $p(x)$-space-bounded for some function $p : \mathbb{N} \to \mathbb{N}$ if for every word w of length n, there exists a run of the Turing machine that does not visit more than $p(n)$ different tape cells. For some easily computable function $p(x)$, we can modify a given $p(x)$-space-bounded Turing machine to never visit more than $p(x)$ tape cells along any of its runs without altering the language of the Turing machine. Intuitively, we do this by first letting our modified Turing machine write tape end markers onto the tape, and then starting the actual computation, while transitioning to a rejecting state from which we never move away once we are about to cross a tape end marker. By extending the tape alphabet accordingly, we can make sure that we do not have to overwrite the initial tape content for doing so.

In addition to non-deterministic Turing machines, there are also *alternating Turing machines*. These

introduce the possibility to combine universal and non-deterministic branching in a single machine. States are labelled by whether they are universally or existentially branching in this case, and we require to have an accepting run tree of the Turing machine for some word to call the word accepted by the Turing machine.

Satisfiability solving In the *satisfiability problem* (*SAT*), we are given a finite set of variables V and a Boolean formula ψ over the variables V, and we want to determine whether there exists a variable valuation to V that satisfies ψ or not. Typically, for the SAT problem, ψ is in *conjunctive normal form*, i.e., we have $\psi = \psi_1 \wedge \ldots \wedge \psi_n$, where for every $i \in \{1, \ldots, n\}$, ψ_n is a *clause* of the form $l_{i,0} \vee \ldots \vee l_{i,k_i}$, where the elements $l_{i,j}$ are *literals*, i.e., of the form $\neg v$ or v for some $v \in V$.

The problem of SAT solving is to check the satisfiability of a Boolean formula. In case of a positive answer, we also want to obtain such a satisfying valuation, i.e., an assignment $f : V \to \mathbb{B}$ to the variables that satisfies the formula.

Complexity classes In the field of computational complexity, we analyse decision problems and categorise them by the worst-case running times and space requirements that any algorithm for a problem under concern can only meet or exceed. The categories that we get in this way form the so-called *complexity classes*. The following list contains some commonly used such classes:

PTIME: the class of all problems that are solvable by a deterministic Turing machine in time (and space) polynomial in the length of the input

NP: the class of all problems that are solvable by a non-deterministic Turing machine in time (and space) polynomial in the length of the input

PSPACE: the class of all problems that are solvable by a deterministic Turing machine in space polynomial in the length of the input

APSPACE: the class of all problems that are solvable by an alternating Turing machine in space polynomial in the length of the input

EXPTIME: the class of all problems that are solvable by a deterministic Turing machine in time (and space) exponential in the length of the input

2EXPTIME: the class of all problems that are solvable by a deterministic Turing machine in time (and space) doubly-exponential in the length of the input

Bibliography

Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

Dana Angluin. Local and global properties in networks of processors (extended abstract). In Raymond E. Miller, Seymour Ginsburg, Walter A. Burkhard, and Richard J. Lipton, editors, *STOC*, pages 82–93. ACM, 1980.

ARM Ltd. Amba™ specification (rev. 2). Available at `www.arm.com`, 1999.

Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Trans. Software Eng.*, 19(11):1015–1027, 1993.

Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar sequential processes. In *POPL*, pages 191–201, 1989.

Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, 1998.

Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2012.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

Bernd Becker, Rüdiger Ehlers, Matthew Lewis, and Paolo Marin. ALLQBF solving by computational learning. In S. Chakraborty and M. Mukund, editors, *10th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 7561 of *Lecture Notes in Computer Science*, pages 370–384. Springer Verlag, 2012.

Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking (extended abstract). In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer, 1994.

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In Rudy Lauwereins and Jan Madsen, editors, *DATE*, pages 1188–1193. ACM, 2007a. doi: 10.1145/1266366.1266622.

Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from PSL. *Electr. Notes Theor. Comput. Sci.*, 190(4):3–16, 2007b. doi: 10.1016/j.entcs.2007.09.004.

Roderick Bloem, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Synthesizing robust systems. In Armin Biere and Carl Pixley, editors, *FMCAD*, pages 85–92. IEEE, 2009.

Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Robustness in the presence of liveness. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2010a.

Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - a new requirements analysis tool with synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010b.

Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012.

Mikolaj Bojańczyk. The common fragment of ACTL and LTL. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 172–185. Springer, 2008.

Patricia Bouyer, Laura Bozzelli, and Fabrice Chevalier. Controller synthesis for MTL specifications. In Christel Baier and Holger Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2006.

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *J. Symb. Log.*, 34(2):166–170, 1969.

Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

Philippe Chatalic and Laurent Simon. Multiresolution for SAT checking. *International Journal on Artificial Intelligence Tools*, 10(4):451–481, 2001.

Krishnendu Chatterjee, Thomas A. Henzinger, and Florian Horn. Finitary winning in omega-regular games. *ACM Trans. Comput. Log.*, 11(1), 2009.

Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll. Mgsyn: Automatic synthesis for industrial automation. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 658–664. Springer, 2012.

S. Chinchali, S.C. Livingston, U. Topcu, J.W. Burdick, and R.M. Murray. Towards formal synthesis of reactive controllers for dexterous robotic manipulation. In *Robotics and Automation (ICRA)*, pages 5183 –5189, 2012. doi: 10.1109/ICRA.2012.6225257.

Alonzo Church. Logic, arithmetic and automata. In *Proc. 1962 Intl. Congr. Math.*, pages 23–25, Upsala, 1962.

Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 1988.

Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 450–462, London, UK, 1993. Springer-Verlag.

Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, pages 427–432, 1995.

Lorenzo Clemente. Büchi automata can have smaller quotients. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2011.

Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3): 201–215, 1960.

Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

Anuj Dawar and Erich Grädel. The descriptive complexity of parity games. In Michael Kaminski and Simone Martini, editors, *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 354–368. Springer, 2008.

Luca de Alfaro and Marco Faella. An accelerated algorithm for 3-color parity games with an application to timed games. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 2007.

Fabrice Derepas and Paul Gastin. Model checking systems of replicated processes with SPIN. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 235–251. Springer, 2001.

Rayna Dimitrova and Bernd Finkbeiner. Synthesis of fault-tolerant distributed systems. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2009.

A. Duret-Lutz. LTL translation improvements in `spot`. In *Proceedings of the Fifth international conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'11, pages 72–83, Swinton, UK, 2011. British Computer Society.

Rüdiger Ehlers. Symbolic bounded synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2010a.

Rüdiger Ehlers. Minimising deterministic Büchi automata precisely using SAT solving. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 326–332. Springer, 2010b. doi: 10.1007/978-3-642-14186-7.

Rüdiger Ehlers. Short witnesses and accepting lassos in ω-automata. In *4th International Conference on Language and Automata Theory and Applications (LATA)*, volume 6031 of *Lecture Notes in Computer Science*, pages 261–272. Springer Verlag, 2010c.

Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer, 2011a.

Rüdiger Ehlers. Experimental aspects of synthesis. In Johannes Reich and Bernd Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 1–16, 2011b.

Rüdiger Ehlers. Small witnesses, accepting lassos and winning strategies in omega-automata and games. Reports of SFB/TR 14 AVACS 80, SFB/TR 14 AVACS, 2011c.

Rüdiger Ehlers. ACTL $\cap$ LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2012a.

Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods in System Design*, 40(2):232–262, 2012b.

Rüdiger Ehlers and Bernd Finkbeiner. On the virtue of patience: Minimizing Büchi automata. In Jaco van de Pol and Michael Weber, editors, *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010. doi: 10.1007/978-3-642-16164-3.

Rüdiger Ehlers and Bernd Finkbeiner. Monitoring realizability. In Sarfraz Khurshid and Koushik Sen, editors, *RV*, volume 7186 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 2011a.

Rüdiger Ehlers and Bernd Finkbeiner. Reactive safety. In Giovanna D'Agostino and Salvatore La Torre, editors, *GandALF*, volume 54 of *EPTCS*, pages 178–191, 2011b.

Rüdiger Ehlers and Daniela Moldovan. Sparse positional strategies for safety games. In Doron Peled and Sven Schewe, editors, *SYNT*, volume 84 of *EPTCS*, pages 1–16, 2012.

Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In *Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)*, pages 91 – 100, 2012a.

Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 91 –100, oct. 2012b.

E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *FOCS*, pages 328–337. IEEE Computer Society, 1988.

E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, pages 368–377. IEEE Computer Society, 1991.

E. Allen Emerson and Kedar S. Namjoshi. On reasoning about rings. *Int. J. Found. Comput. Sci.*, 14(4): 527–550, 2003.

E. Allen Emerson and Jai Srinivasan. A decidable temporal logic to reason about many processes. In *PODC*, pages 233–246, 1990.

E. Allen Emerson, Tom Sadler, and Jai Srinivasan. Efficient temporal reasoning. In *POPL*, pages 166–178, 1989.

Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer, 2000.

Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *ICALP*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707. Springer, 2001.

Faith E. Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2009.

Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Compositional algorithms for LTL synthesis. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *ATVA*, volume 6252 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2010.

N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.

Bernd Finkbeiner. Personal communication, 2010.

Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *LICS*, pages 321–330. IEEE Computer Society, 2005. doi: 10.1109/LICS.2005.53.

Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Automated Formal Methods (AFM)*, 2007.

Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

Riccardo Forth and Paul Molitor. An efficient heuristic for state encoding minimizing the BDD representations of the transistion relations of finite state machines. In *ASP-DAC*, pages 61–66. ACM, 2000.

Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1): 98–115, 1987. doi: 10.1145/7531.7919.

Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *CIAA*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2003.

G. Frobenius and L. Stickelberger. Über Gruppen von vertauschbaren Elementen. *J. Reine und Angew. Math.*, 86:217–262, 1878.

Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3):215–237, 2009.

Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. Synthesis of AMBA AHB from formal specification: a case study. *International Journal on Software Tools for Technology Transfer*, 2011. doi: 10.1007/s10009-011-0207-9.

Wilsin Gosti, Tiziano Villa, Alexander Saldanha, and Alberto L. Sangiovanni-Vincentelli. FSM encoding for BDD representations. *Applied Mathematics and Computer Science*, 17(1):113–124, 2007. doi: 10.2478/v10006-007-0011-6.

Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*, 2002. Springer.

Sankar Gurumurthy, Roderick Bloem, and Fabio Somenzi. Fair simulation minimization. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 610–624. Springer, 2002.

Anthony Hall. Realising the benefits of formal methods. *J. UCS*, 13(5):669–678, 2007.

Kosaburo Hashiguchi. Representation theorems on regular languages. *J. Comput. Syst. Sci.*, 27(1): 101–115, 1983.

Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to Uppaal. In Kim Guldstrand Larsen and Peter Niebert, editors, *FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.

Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2006.

C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. In *ICCD*, pages 230–234, 1993.

C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

Swen Jacobs and Roderick Bloem. Parameterized synthesis. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2012.

David Janin and Giacomo Lenzi. On the relationship between monadic and weak monadic second order logic on arbitrary trees, with applications to the mu-calculus. *Fundam. Inform.*, 61(3-4):247–265, 2004.

Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *PODC*, pages 13–22, New York, NY, USA, 1985. ACM. doi: 10.1145/323596.323598.

Joachim Klein and Christel Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theor. Comput. Sci.*, 363(2):182–195, 2006. doi: 10.1016/j.tcs.2006.07.022.

Uri Klein and Amir Pnueli. Revisiting synthesis of GR(1) specifications. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Haifa Verification Conference (HVC)*, volume 6504 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2010.

Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In Armin Biere and Carl Pixley, editors, *FMCAD*, pages 152–159. IEEE, 2009.

Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging unrealizable specifications with model-based diagnosis. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2010.

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *ICRA*, pages 3116–3121. IEEE, 2007.

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25:1370–1381, 2009.

Sriram C. Krishnan, Anuj Puri, Robert K. Brayton, and Pravin Varaiya. The Rabin index and chain automata, with applications to automatas and games. In Pierre Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 253–266. Springer, 1995.

James H. Kukula and Thomas R. Shiple. Building circuits from relations. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 113–123. Springer, 2000.

O. Kupferman and M.Y. Vardi. Synthesis with incomplete informatio. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997.

Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *LICS*, pages 389–398. IEEE Computer Society, 2001a.

Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001b.

Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542. IEEE, 2005.

Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

Orna Kupferman, Gila Morgenstern, and Aniello Murano. Typeness for omega-regular automata. *Int. J. Found. Comput. Sci.*, 17(4):869–884, 2006a.

Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006b.

Orna Kupferman, Shmuel Safra, and Moshe Y. Vardi. Relating word and tree automata. *Ann. Pure Appl. Logic*, 138(1-3):126–146, 2006c.

Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 133–138, New York, NY, USA, 1981. ACM. doi: 10.1145/567532.567547.

Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 43–50. IEEE, 2011.

Parthasarathy Madhusudan. Synthesizing reactive programs. In Marc Bezem, editor, *CSL*, volume 12 of *LIPIcs*, pages 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

Parthasarathy Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2002.

Monika Maidl. The common fragment of CTL and LTL. In *FOCS*, pages 643–652, 2000.

Donald A. Martin. Borel determinacy. *The Annals of Mathematics*, 102(2):363–371, 1975.

Donald A. Martin. A purely inductive proof of Borel determinacy. In *Recursion theory*, volume 42 of *Symposium on Pure Mathematics*, pages 303–308. American Mathematical Society, 1982.

Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, pages 272–277, 1993.

Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.

Andreas Morgenstern. *Symbolic Controller Synthesis for LTL Specifications*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, 2010.

Andreas Morgenstern and Klaus Schneider. A LTL fragment for GR(1)-synthesis. In Johannes Reich and Bernd Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 33–45, 2011.

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. *Theor. Comput. Sci.*, 141(1&2):69–107, 1995.

Damian Niwinski and Igor Walukiewicz. Relating hierarchies of word and tree automata. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *STACS*, volume 1373 of *Lecture Notes in Computer Science*, pages 320–331. Springer, 1998.

Necmiye Ozay, Ufuk Topcu, Richard M. Murray, and Tichakorn Wongpiromsarn. Distributed synthesis of control protocols for smart camera networks. In *IEEE/ACM Second International Conference on Cyber-Physical Systems (ICCPS)*, pages 45–54, Washington, DC, USA, 2011. IEEE Computer Society. doi: 10.1109/ICCPS.2011.22.

Radek Pelánek and Jan Strejcek. Deeper connections between LTL and alternating automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *CIAA*, volume 3845 of *Lecture Notes in Computer Science*, pages 238–249. Springer, 2005.

Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 649–655. Springer, 2011.

Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. *Logical Methods in Computer Science*, 3(3), 2007.

Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer, 1989a.

Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989b.

Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, pages 746–757. IEEE Computer Society, 1990.

Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.

Michael O. Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, 1972.

Vasumathi Raman and Hadas Kress-Gazit. Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 663–668. Springer, 2011.

Desh Ranjan, Richard Chang, and Juris Hartmanis. Space bounded computations: review and new separation results. *Theoretical Computer Science*, 80(2):289 – 302, 1991. doi: 10.1016/0304-3975(91)90391-E.

Shmuel Safra. On the complexity of ω-automata. In *FOCS*, pages 319–327. IEEE Computer Society, 1988.

Shmuel Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, March 1989.

Karem A. Sakallah. *Handbook of Satisfiability*, chapter Symmetry and Satisfiability, pages 289–338. In Biere et al. (2009), 2009.

Sven Schewe. *Synthesis of Distributed Systems*. PhD thesis, Saarland University, 2008.

Sven Schewe. Beyond hyper-minimisation—minimising DBAs and DPAs is NP-complete. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 8 of *LIPIcs*, pages 400–411. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2007.

Joel I. Seiferas, Michael J. Fischer, and Albert R. Meyer. Separating nondeterministic time complexity classes. *J. ACM*, 25(1):146–167, 1978.

E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html`.

A. Prasad Sistla and Patrice Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004. doi: 10.1145/1011508.1011511.

Saqib Sohail and Fabio Somenzi. Safety first: A two-stage algorithm for LTL games. In Armin Biere and Carl Pixley, editors, *FMCAD*, pages 77–84. IEEE, 2009. doi: 10.1109/FMCAD.2009.5351138.

Saqib Sohail, Fabio Somenzi, and Kavita Ravi. A hybrid algorithm for LTL games. In Francesco Logozzo, Doron Peled, and Lenore D. Zuck, editors, *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2008.

Wolfgang Thomas. *Handbook of Theoretical Computer Science – Vol. B: Formal Models and Semantics*, chapter Automata on Infinite Objects, pages 133–191. MIT Press, 1994.

Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*. Springer, 1995.

Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In William McCune, editor, *CADE*, volume 1249 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

Moshe Y. Vardi and Larry J. Stockmeyer. Improved upper and lower bounds for modal logics of programs: Preliminary report. In *STOC*, pages 240–251. ACM, 1985.

Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.

Igor Walukiewicz. Synthesis: Words and traces. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *ATVA*, volume 6252 of *Lecture Notes in Computer Science*, pages 18–21. Springer, 2010.

Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, 2000.

Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In Karl Henrik Johansson and Wang Yi, editors, *HSCC*, pages 101–110. ACM ACM, 2010a.

Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Automatic synthesis of robust embedded control software. In *AAAI Spring Symposium on Embedded Reasoning*, 2010b.

Huan Xu, Ufuk Topcu, and Richard M. Murray. Reactive protocols for aircraft electric power distribution. In *Conference on Decision and Control*, 2012.

Index