Parameterized Synthesis of Guarded Systems

Simon Außerlechner

Simon Außerlechner

Parameterized Synthesis of Guarded Systems

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Advisor: Univ.-Prof. M.Sc. Ph.D. Roderick Bloem

Co-Advisor: Dipl.-Inform. Dr.-Ing. Swen Jacobs

Institute of Applied Information Processing and Communications (IAIK)
Graz University of Technology
8010 Graz, Austria

Graz, May 2015

Abstract

Ensuring the correctness of distributed systems is a driving force behind the development of formal methods. In contrast to informal methods like testing, verification guarantees to reveal hard-to-find bugs, while synthesis even allows to create systems that are correct by construction. Distributed systems are often parameterized regarding the number of processes, that is, they consist of a variable number of processes that are either isomorphic or belong to a finite group of isomorphic processes. This scalability makes known formal methods undecidable in general, since correctness needs to be shown for each of the possibly infinitely many parameterized system instantiations with a particular size.

Nevertheless, research revealed several methods which allow to efficiently solve the Parameterized Model Checking Problem (PMCP) and the Parameterized Synthesis Problem for certain system classes, respectively. This thesis aims at establishing efficient Parameterized Synthesis for the class of guarded systems. By reducing the model checking of parameterized guarded systems to the model checking of several small-sized guarded systems, Emerson and Kahlon (2000) showed that the PMCP for this class is efficiently decidable for special types of CTL*\X specifications. In order to provide efficient Parameterized Synthesis, we lift these results to the synthesis domain. One major problem of the already existing results is that they only apply to closed systems and do not consider fairness. However, the usual goal of synthesis is to find an open system implementation under the aspect of fairness. For this reason, we first develop an approach that directly uses the existing cutoffs for synthesizing open systems. In order to achieve efficient synthesis under consideration of fairness, we then revisit and extend the result of Emerson and Kahlon. Furthermore, we describe our implementation that solves the Bounded Synthesis Problem for guarded systems. Based on our prototype, we finally evaluate the Parameterized Synthesis of guarded systems using the newly obtained cutoffs.

Kurzfassung

Die Sicherstellung der Korrektheit verteilter Systeme ist eine treibende Kraft für die Entwicklung formaler Methoden. Im Gegensatz zu informalen Methoden, wie beispielsweise Testen, garantiert die Verifikation, dass schwer zu findende Fehler gefunden werden, während aus der Synthese per Definition korrekte Systeme resultieren. Oft sind verteilte Systeme bezüglich der Prozessanzahl parametrisiert, das heißt sie bestehen aus einer variablen Anzahl von Prozessen, welche entweder isomorph sind oder einer Gruppe isomorpher Prozesse angehören. Bekannte formale Methoden sind aufgrund dieser Skalierbarkeit generell unentscheidbar, da die Korrektheit für jede der unendlich vielen verschieden großen Systeme gezeigt werden muss.

Nichtsdestotrotz wurden mehrere Ansätze entwickelt, die erlauben, das Parameterized Model Checking Problem (PMCP) sowie das Parameterized Synthesis Problem für gewisse Systemklassen effizient zu lösen. Diese Arbeit beschäftigt sich mit der effizienten parametrisierten Synthese für die Klasse der Guarded Systems. Emerson und Kahlon (2000) haben gezeigt, dass das PMCP für diese Systemklasse unter Betrachtung eingeschränkter CTL*\X-Spezifikationen durch Model Checking einer finiten Anzahl kleiner Systeminstanzen effizient entscheidbar ist. Um zu zeigen, dass auch die parametrisierte Synthese für diese Systemklasse effizient entscheidbar ist, heben wir diese Resultate von der Model-Checking- in die Synthese-Domäne. Ein Hauptproblem dabei ist, dass die vorhandenen Resultate nur für geschlossene Systeme gültig sind und Fairness nicht unterstützen. Jedoch ist das Ziel von Synthese hauptsächlich die Konstruktion offener Systeme unter Annahme fairer Environments. Zunächst entwickeln wir daher einen Ansatz zur Synthese offener Systeme, welcher die bereits vorhandenen Resultate verwendet. Um auch die Synthese von fairen, offenen Systemen zu ermöglichen, erweitern wir anschließend die Cutoff-Resultate von Emerson und Kahlon um solche, welche Fairness unterstützen. Weiter beschreiben wir sowohl unsere Implementierung, welche das Bounded Synthesis Problem für Guarded Systems löst, als auch das zugrunde liegende SMT-Encoding. Basierend auf unserem Prototypen evaluieren wir schließlich die Parameterized Synthesis von Guarded Systems unter Verwendung der neuen Cutoffs.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present diploma thesis.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Diplomarbeit identisch.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

Acknowledgements

There are many people who made this thesis possible and who I am indebted to. First and foremost, I want to thank my advisors Prof. Roderick Bloem, Swen Jacobs, and Ayrat Khalimov for their enthusiasm, their support, and their patience during the countless hours on the blackboard. Without you, I would not have been able to finish this thesis.

Second, I want to thank my study colleagues for the hours spent together in preparation of this final effort. Apart from that I express my gratitude to Paul Schwann and Nikolaos Korkakakis at NXP for their understanding and encouragement concerning my thesis.

Furthermore, I thank my former school teacher Christoph Schönherr as well as Prof. Karl-Christian Posch and Prof. Franz Wotawa for inspiration during my school and study time respectively. Last but not least, I want to thank my parents and friends, my uncles and aunties, and my girlfriend.

Simon Außerlechner
Graz, Austria, May 2015

Credits

I would like to thank the following individuals and organizations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [4].
- Ayrat Khalimov et al. [51] made their great parameterized synthesis tool PARTY publicly available and allowed us to reuse their source code for our prototype implementation.

Chapter 1

Introduction

1.1 Background and Motivation

Today we rely on computer systems in many fields of application. Especially in safety-critical areas like medicine, transportation, aeronautics, astronautics, and so forth, faults in hardware and software have dramatic consequences, ranging from millions of dollars worth of damage to loss of human life. [75] In order to assure a constantly high product quality and minimize risks related to the development process, undesired program behaviour must be detected as soon as possible. For ensuring that a system is correct, developers rely on informal methods like extensive testing and simulation, but also on formal methods like verification, where the system is proved to satisfy the specification. Synthesis even goes a step further than verification, and aims at creating systems that are correct by construction, making risky and cost-intensive manual development superfluous.

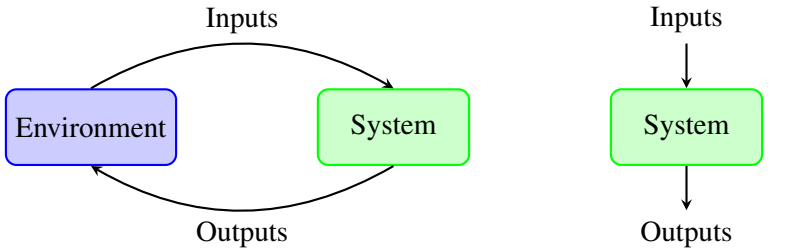
A plethora of real-world systems consist of multiple autonomous entities that constantly react on some input (e.g., a button press), and interact with each other by exchanging some information. In some of these so-called *distributed reactive systems*, entities are allowed to act (i.e., change their state) independently from all other participants, and synchronize with them at certain points of time. The entities' concurrency makes testing distributed systems very challenging, because the occurrence of some undesired behaviour in one process can strongly depend on the timing with respect to the overall system. Thus, reproducing certain bugs is very hard, and the outcome of tests is not predictable: If the timing during the test execution is such that the bug is revealed, the test fails, otherwise the test succeeds although the system is not correct. By contrast, verification is unconditionally capable of revealing the presence of such race conditions, while synthesis guarantees their absence if properly described in the specification.

Both recent advances in the field of formal methods and increasing computational power caused formal methods to become feasible for industrial purposes. Especially the use of model checking, an algorithmic verification approach, became widespread in industry because it allows to do a correctness check in a fully automated manner [76]. Nevertheless, both the model checking problem and the synthesis problem cannot yet be solved efficiently for all kind of systems. In particular, applying formal methods to distributed systems of an arbitrary size has become a challenging field of research.

1.1.1 Distributed Reactive Systems

A *reactive system* is a permanently operating system which continually reacts to inputs provided by an external entity (see Figure 1.1a), i.e., the environment [46]. By contrast, a *transformational system* (see Figure 1.1b) can be represented as a mapping from an input value to an output value [60]. Reactive systems are often *distributed*, that is, they involve multiple processes. Depending on the structure of the system, processes are allowed to run in parallel. Each process possibly communicates with the environment, but also exchanges information with other processes in order to achieve overall consistency (e.g.,

synchronization). Reasons for using distributed systems are speed, fault-tolerance as well as physical distribution requirements [45].



(a) Reactive System

(b) Transformational System

Figure 1.1: Structure of reactive and transformational system

1.1.2 Model Checking

The growing interest for the verification of concurrent reactive systems arises from the fact that testing is not capable of revealing timing-dependent bugs with certainty. Before the invention of model checking, most approaches were either based on theorem proving or exhaustive state search [18]. Both suffer from lacking scalability. On the one hand, theorem proving requires human intervention. This makes proving large systems a time-consuming process. On the other hand, applying exhaustive state search to large systems is infeasible due to the number of possible states. In the 1970s, Burgstall [15], Kröger [54], Pnueli [64], and others proposed using Temporal Logics for proving programs. Originally invented by linguists and philosophers, Temporal Logics allow to describe the ordering of events without explicitly introducing the aspect of time. We distinguish between Linear Time Logics, e.g., LTL, and Branching Time Logics, e.g., CTL (Computational Tree Logic). Pnueli [64] also demonstrated that Temporal Logics allow to express properties of concurrent programs in an elegant way. Clarke and Emerson combined the idea of using Temporal Logic specifications and efficient state exploration, and developed model checking, an efficient algorithmic approach that allows to decide if a system satisfies a formal specification in CTL. Clarke [18] points out that the word “model” does not refer to the fact that an abstraction of the system is checked, but instead has its origin in the purpose of the approach, i.e., to check whether a system is a model for the specification.

In contrast to verification techniques like theorem proving, model checking is fully automatic and does not require the user to manually construct proofs. Additionally, if the model checking algorithm finds a state that violates the specification, it is able to construct a counterexample, which helps the developer to better identify the causes of the violation. Moreover, model checking can be applied to specifications of partial correctness. For example, model checking can be used for checking only safety properties (e.g. mutual exclusion). Both LTL and CTL* (a Temporal Logic which unites the expressiveness of LTL and CTL) model checking have been shown to be PSPACE-complete with respect to the size of the specification [70, 20, 21]. The model checking problem for distributed systems can be solved by considering the distributed system as a single centralized system.

A major drawback of model checking is that it suffers from a so-called state explosion, making the approach inefficient when checking systems with a large number of states, including distributed systems with a large number of entities. Improvements like Symbolic Model Checking [14, 61] and Bounded Model Checking [12], partial order reduction [74, 42] etc. made it possible to check complex systems whose state space is orders of magnitudes larger than the state space of systems which could be checked with the original algorithm.

Furthermore, there exist techniques that reduce the complexity for distributed systems model check-

ing. These techniques include, e.g., compositional reasoning, which allows to verify the particular components in isolation [10], abstraction, which reduces the complexity by modelling simple (e.g. arithmetic) relations [22], and symmetry reduction [47], which is based on the observation that many distributed systems contain many identical processes.

1.1.3 Synthesis

The synthesis problem is to find a system that satisfies a given formal specification. In contrast to verification, synthesis does not check whether an existing system is correct, but constructs a system that is correct with respect to the specification. Thus, synthesis automates the step of implementing the system, changing the system development process from programming systems to defining the desired systems' properties. A detailed specification which fully describes the behaviour of the desired system is a prerequisite for synthesis.

The problem of determining whether there exists a system that satisfies the specification is the *realizability problem*. The realizability problem and the synthesis problem were first formulated by Church [17]. Rabin [67] as well as Büchi and Landweber [13] solved the synthesis problem for specifications written in the monadic second-order logic of one successor (S1S) in two different ways. While Rabin's solution is based on infinite trees, Büchi and Landweber developed an algorithm that is based on finding a winning strategy for infinite games. Synthesis of specifications in LTL [64] was introduced by Pnueli and Rosner [65], who provided a synthesis algorithm and showed that the problem is 2EXPTIME-complete regarding the specification. Their algorithm with double-exponential runtime consists of the following steps. First, the specification is converted into a non-deterministic Büchi automaton. This conversion yields a single exponential blow-up of the state space. Second, Safra's determinization algorithm is applied in order to retrieve a deterministic Rabin automaton with a number of states that is double-exponential regarding the size of the original specification. Finally, an emptiness check is applied to the tree automaton and the infinite tree that represents the implementation. Kupferman and Vardi [57] point out that Safra's construction is difficult to implement, and propose an alternative approach that goes without the determinization step. In this algorithm the LTL formula is first translated into a universal co-Büchi tree automaton, and then converted into a non-deterministic parity tree automaton. This approach is simpler than the Safra construction, and thus easier to implement. Moreover, the improved "Safraless" translation enables BDD-based implementations, and also provides support for optimizations. [55]

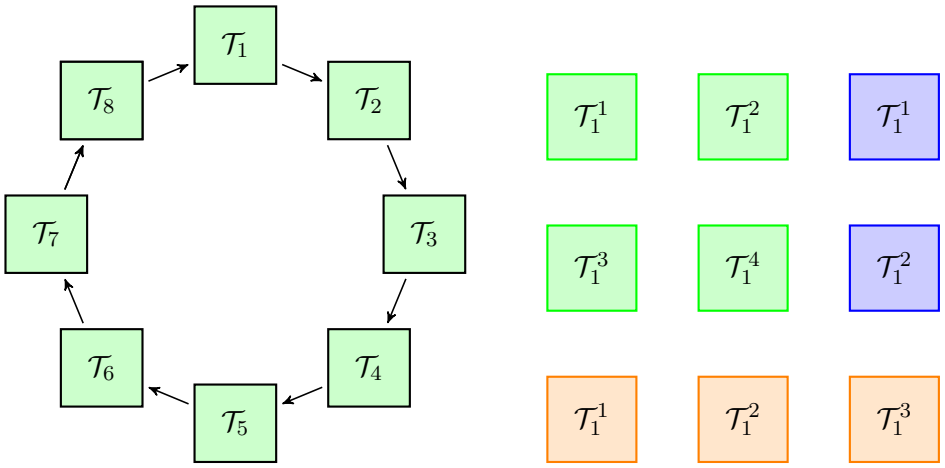
The theoretical lower bound for the synthesis of LTL specifications is double exponential. Nevertheless, there exist certain fragments within LTL that are sufficient for real-world specifications, and can be efficiently synthesized. One popular example of such an LTL fragment is GR(1), for which Piterman et al. provide a polynomial synthesis algorithm [63]. A further challenge is the synthesis of distributed systems. As already mentioned, the model checking problem is PSPACE-complete for both centralized and distributed systems. However, Pnueli and Rosner proved that the synthesis of systems with more than one entity (i.e., process) is undecidable in general, and non-elementary decidable for systems with hierarchical architectures. As in the general case of synthesis, there exist approaches, which efficiently find solutions for specific subsets of synthesis problems [53, 16].

Finkbeiner and Schewe introduced the concept of information forks, and showed that architectures with such information forks are undecidable [37]. One major aspect of synthesis is that the size of the implementation is left implicit, whereas it is explicitly provided in case of model checking [69]. Finkbeiner and Schewe observed that many specifications are satisfied by systems with a small number of states. Using this observation, they proposed a semi-algorithm to solve the synthesis problem [69]. Their bounded synthesis algorithm consists of the following steps. First, the maximum size of the desired implementation is restricted a priori. Second, the specification is converted into a universal co-Büchi Tree Automaton (UCT). Then, the language emptiness check of the UCT is reduced to the problem of assigning a label to each state of the run graph of the UCT on the implementation. Finally, the annotation is encoded in SMT, allowing to describe the synthesis problem as an SMT problem. If the

SMT problem is satisfiable, a valid system is found, otherwise the maximum system size is increased, followed by solving the bounded synthesis problem for the new upper bound. Using an SMT solver entails greater flexibility compared to a purely automaton-based emptiness check. While it is difficult or even impossible to encode additional system requirements into an automaton, adding corresponding SMT constraints is simple, which is especially favourable in case of distributed synthesis.

1.2 Problems Addressed in this Thesis

Many real-world distributed systems are based on a structure that enables scalability regarding the system size. For example, consider bus protocols, which support any number of slaves, or cache coherence protocols, which are designed to ensure consistency of an arbitrary number of caches. The structure of these systems is also called *architecture*. It defines the interface (inputs and outputs) of each process as well as the communication between processes. In case of arbitrarily scalable systems, we usually assume that all processes are isomorphic (they share the same implementation) or that there is a finite number of isomorphic process groups. For example, consider a token ring network of isomorphic processes. The system is arbitrarily scalable because it can be extended by inserting an additional entity and connecting it with its neighbours (see Figure 1.2a). If a process is removed, the two previous neighbour processes are connected with each other.



(a) Token Ring Architecture **(b)** Guarded Systems Architecture

Figure 1.2: Distributed systems based on parameterized architectures

A *parameterized architecture* is a family of architectures which only vary in the number of processes. Given implementations of system processes, the goal of verifying a parameterized architecture against a specification is to check whether the specification holds for all system sizes. Parameterized architecture synthesis (abbreviated to *parameterized synthesis*) aims at constructing a parameterized architecture that is correct for each system size. Suzuki [72] showed that the model checking problem for parameterized architectures with respect to a fixed implementation is undecidable independent from the specification language. However, efficient model checking against special specification types is possible for some parameterized architectures. This is possible because of the existence of small-sized systems which cover all possible behaviours of any larger system regarding the particular specification. Therefore, only a finite number of small systems needs to be considered.

In this thesis, we consider the parameterized architecture of guarded systems (see Figure 1.2b). Originally described by Clarke and Emerson [19], guarded systems consist of multiple groups of isomorphic processes, all based on a set of so-called process templates. Processes communicate to each other by reporting their current state. Each process' behaviour is influenced by the states of all other processes.

Emerson and Kahlon [35] describe how model checking of guarded systems against three different types of restricted CTL* specifications can be efficiently realized.

This thesis aims at enabling efficient synthesis for guarded systems. We consider the cutoffs proved by Emerson and Kahlon [35] for the purpose of model checking guarded systems, and aim at lifting their results to synthesis. Two main problems need to be tackled. First, the results of Emerson and Kahlon are for closed systems, that is, systems which cannot be influenced by some external entity. For our purpose, this restriction is insufficient, since the majority of real-world distributed systems expect some stimuli. For example, a memory cache provides a signal which allows a control unit to trigger a write action, while a bus arbiter has some input which is used by an external entity to signal a bus request.

The second problem to be solved is that the specification types from [35] do not allow to express fairness. In the context of synthesis, this possibly results either in unrealizability (because the specification is too strong) or undesired trivial solutions for the implementation (because the specification is too weak). To tackle this problem, we need to extend the specification by so-called fairness constraints — formulae which ensure that the system only needs to satisfy the specification in fair cases. A further problem arising from introducing fairness constraints is that the extended specifications are not compatible with the specification types from [35]. As a result, we cannot use the existing cutoffs for the synthesis under consideration of fairness.

1.3 Outline of the Solution

We pursue two different approaches. The first approach is to directly use the cutoff results of Emerson and Kahlon [35] for synthesis. Here, we synthesize a closed system which is finally converted into an open system. To this end, we introduce a set of conversion rules, which allow to convert each arbitrary open system into a corresponding closed system and vice versa. However, this approach has two major drawbacks. On the one hand, the conversion yields an exponential blow-up of the state space. On the other hand, fairness is not considered.

In the second approach, we show that the existing cutoff results can be used for open systems directly. Moreover, we amend the existing cutoff results for guarded systems under the aspect of fairness. In particular, we introduce fairness constraints such that unfair behaviour regarding the environment does not cause trivial synthesis results. Moreover, we focus on introducing cutoffs that allow us to detect whether there exists some large system where at least one process or even the whole system deadlocks.

After introducing the new cutoff results, we modify the semi-decision bounded synthesis algorithm [69] in order to synthesize guarded systems. Finally, we develop a prototype which implements this algorithm, and evaluate synthesis results for a set of small examples.

1.4 Structure of this Thesis

This document is organized as follows. In Chapter 2 we give an overview of related work in the field of module checking (model checking of open systems), parameterized model checking, and parameterized synthesis. We introduce the preliminaries in Chapter 3, and define the considered system model in Chapter 4. Chapter 5 summarizes the work of Emerson and Kahlon [35]. We describe the theoretical aspects of parameterized synthesis for guarded systems in Chapter 6. Chapter 7 deals with the SMT-based encoding of the bounded synthesis algorithm for guarded system synthesis in SMT. In Chapter 8 we describe our prototype implementation and practical experiments. Chapter 9 concludes our work with a discussion of our results and an overview about future work.

Chapter 2

Related Work

2.1 Model Checking Open Systems

Kupferman and Vardi [58] investigate the complexity of model checking open systems (*module checking*) against specifications in linear and branching time logics. They show that CTL module checking is EXPTIME-complete, whereas CTL* module checking is 2EXPTIME-complete. Moreover, the module checking problem is harder (PTIME-complete) than model checking (NLOGSPACE) with respect to the program complexity, i.e., the size of the implementation (under consideration of a fixed specification) [58]. However, both model checking and module checking for universally quantified temporal logics are in PSPACE with respect to the specification, and in NLOGSPACE with respect to the system size. Kupferman and Vardi [56] further consider systems with *incomplete information*, i.e., systems with variables that are not visible for the environment. For universal temporal logics, the complexity of module checking systems with incomplete information is equivalent to the complexity of the standard module checking problem. For non-universal temporal logics, the module checking problem is harder under the aspect of incomplete information.

2.2 Parameterized Model Checking

Apt and Kozen [5] show that the Parameterized Model Checking Problem (PMCP) is undecidable in general. Undecidability even holds for ring-based systems with isomorphic processes [72]. As described by Arons et al. [6], the research community follows two approaches in order to tackle the undecidability problem. On the one hand, research focuses on identifying fragments for which the PMCP can be reduced to the standard model checking problem. On the other hand, heuristic-based solutions for the PMCP are developed. In the following, we consider publications that are based on the first approach. The PMCP is well studied for systems classes that use some sort of token as a communication primitive. Emerson and Namjoshi [34] prove that the PMCP is decidable for unidirectional token-passing rings of isomorphic processes and special properties in prenex indexed CTL*\X. To this end, they show that the PMCP for these cases is equal to the model-checking problem for systems up to a small number of processes (cutoff). In the considered system model, tokens are only used for signaling, and do not carry values [32]. In [32], Emerson and Kahlon consider the PMCP for the more general system model of message-passing bidirectional token rings with isomorphic processes and LTL\X properties. There exist cutoffs for such systems if the number of token value changes is bounded. Moreover, Emerson and Kahlon provide cutoffs for unidirectional message-passing rings with multiple tokens and a bounded number of value changes. Clarke et al. [24] consider token-passing systems over arbitrary graphs consisting of isomorphic processes that are not direction-aware. For LTL\X properties, they propose a decomposition approach, where a large network graph is split into a finite set of small (constant-sized) networks,

which can be model-checked efficiently. There does not exist such a decomposition for indexed CTL\X specifications [24].

As proved by Aminof et al. [3], the PMCP is generally undecidable for arbitrary topologies and both k-indexed LTL\X and CTL\X properties if processes are direction-aware, that is, each process is allowed to choose at least the token sending or receiving direction. For topologies without direction-aware processes, there exist cutoffs for arbitrary topologies and k-indexed properties in $\text{CTL}^\star_d\text{\X}$, where k is the number of process quantifiers and d is the path quantifiers' nesting depth [3]. Although this result implies that there exists a cutoff for each full-indexed $\text{CTL}^\star_d\text{\X}$ formula, the existence of an algorithm for computing the particular cutoff is not guaranteed. Indeed, there exist certain topologies, for which the PMCP is undecidable regarding prenex indexed LTL\X and CTL\X specifications.

In [35], Emerson and Kahlon show decidability for the PMCP for disjunctive and conjunctive guarded synchronization skeletons [19] and prenex indexed $\text{CTL}^\star\text{\X}$ properties by providing cutoffs. Emerson and Kahlon also consider the PMCP of various cache coherence protocol types [31] and restricted $\text{CTL}^\star$ properties. To this end, they introduce a framework consisting of restricted system types. General snoopy based protocols are modeled as *guarded broadcast protocols*, while invalidation-based snoopy protocols are modeled as *initialized broadcast protocols*. Safety properties of parameterized protocols modeled as *guarded broadcast protocols* are model-checked using a so-called *abstract history graph*. This graph abstracts the system's global state, yielding a state space that is independent from the number of processes in the system. For the more restricted model called *initialized broadcast protocols*, efficient PMCP is established by the existence of a constant cutoff. Moreover, Emerson and Kahlon describe a reduction of the PMCP for directory based protocols to the PMCP for snoopy protocols.

Emerson and Kahlon examine the PMCP for action-based systems under consideration of different communication primitives in systems with a single control process C and an arbitrary number of "user processes" U [30]. These include conjunctive and disjunctive Boolean guards, pairwise and asynchronous rendezvous as well as broadcast actions. The considered specification types are (p1) properties over C in LTL\X, (p2) properties over C in LTL, (p3) regular properties given as regular automata, and (p4) ω-regular properties given as ω-regular automata. Table 2.1 summarizes the results presented in [30]. Note that LTL\X is less expressive than LTL, and LTL is less expressive than ω-regular automata. In the context of action-based systems, disjunctive guards and pairwise rendezvous are equally expressive, and strictly less expressive than asynchronous rendezvous. Hence, the model checking problem for systems with pairwise rendezvous can be solved by model checking the corresponding disjunctive guarded systems (where cutoffs for the PMCP are known [35]). Thus, the PMCP under consideration of LTL\X properties over the control process C is decidable for systems with pairwise rendezvous. By contrast, the PMCP for properties of type 1 are undecidable for asynchronous rendezvous and broadcast actions. The PMCP for type 2 and type 4 properties is decidable for disjunctive guards and pairwise rendezvous, and undecidable for all other communication primitives. For type 3 properties, the PMCP is undecidable for conjunctive guards, and decidable for the other communication primitives. Moreover, Emerson and Kahlon consider systems that use combination of protocols. They show that combinations including conjunctive guards yield undecidability, whereas the PMCP for systems with broadcast actions and asynchronous rendezvous is decidable for type 3 properties. Emerson and Kahlon restrict the model of conjunctive guarded systems based on observations of cache coherence protocols. On the one hand, they require a conjunctive guarded system to be initializable, that is, each state has an unguarded transition to the initial state. On the other hand, transitions must not be blocked by any process which is in its initial state. In other words, each guard must include all initial states. These restrictions yield decidability for the PMCP for systems with conjunctive guards and any other communication primitive under consideration of regular properties (see Column 1 in Table 2.1). The following recent publications consider the cutoff results from [35]. Aminof et al. [2] reduced pairwise rendezvous to disjunctive guards. This result implies decidability for the PMCP of systems with pairwise rendezvous synchronization (by the cutoffs for disjunctive guarded systems from [35]). Spalazzi and Spegni [71] revisit the cutoffs for conjunctive guarded systems to the enable efficient model checking of timed automata.

	Restricted Conjunctive Guards	Disjunctive Guards	Pairwise Rendezvous	Asynchronous Rendezvous	Broadcast
p1	✓ [35]	✓ [35]	✓ [41]	✗ [30]	✗ [30]
p2	✗ [30]	✓ [30]	✓ [30]	✗ (by p1)	✗ (by p1)
p3	✗	✓ [30]	✓ [30]	✓ [28]	✓ [36]
p4	✗ (by p2)	✓ [30]	✓ [30]	✗ (by p2)	✗ [36]

Table 2.1: Decidability of the PMCP for action-based systems (✓ is decidable, ✗ is undecidable)

2.3 Parameterized Synthesis

Undecidability for the general case of distributed synthesis is shown by Pnueli and Rosner [66]. In [48], Jacobs and Bloem show that the parameterized synthesis problem for uni-directional token rings with isomorphic process implementations and LTL\X properties is undecidable. They provide a semi-decision procedure based on bounded synthesis [37] that allows to synthesize arbitrary token-passing networks (without direction awareness) given LTL\X specifications. To this end, they apply the PMCP results of Emerson and Namjoshi [33] (for unidirectional token-rings) and Clarke et al. [24] (for other token-passing networks) to synthesis. Furthermore, Jacobs and Bloem [49] provide a synthesis framework that describes how to obtain a semi-decision synthesis algorithm by lifting verification algorithms for particular system classes to synthesis. The synthesis framework considers different types of PMCP results. Cutoffs which only depend on the architecture and on the specification (*static structure-independent cutoffs*) as well as cutoffs which additionally depend on the size of the implementation (*static structure-dependent cutoffs*) can be applied to parameterized synthesis directly. For static structure-independent cutoffs, the resulting semi-algorithm for synthesis consists of the following steps. First, the cutoff is calculated. Second, the bounded synthesis problem is encoded in SMT for system sizes (number of processes) up to the determined cutoff. Finally, solving the SMT problem yields an implementation in case of satisfiability, otherwise, the bounded synthesis problem is solved for an increased number of states. Synthesis under consideration of static structure-dependent cutoffs is similar. However, due to the cutoffs' dependence on the process size, the considered system size changes in each loop iteration. Therefore, the first step in each loop iteration is computing the current cutoff. *Dynamic cutoffs* are not computed based on syntactic properties of the architecture, the specification, or the implementation [49], but instead require an exploration of the actual implementation. Such cutoffs cannot be lifted to synthesis easily because of their dependence on the implementation, which is given in case of verification, however, a priori unknown in case of synthesis. Further methods for parameterized verification described in the synthesis framework, i.e., induction-based approaches, abstraction-based approaches, and regular model checking, have not been considered for parameterized synthesis so far.

Khalimov et al. [52] extend and improve the approach introduced in [48] considering two different aspects. On the one hand, the original SMT encoding is optimized. For example, the original top-down encoding [37] is replaced by a more compact bottom-up encoding. On the other hand, general optimizations are described. These include using incremental SMT solving, modular synthesis, and specification strengthening, a technique which is sound, but not complete. In [51], Khalimov et al. present PARTY, an implementation of the bounded synthesis approach that is capable of synthesizing monolithic as well as parameterized systems. Alur et al. [73] propose a semi-automatic synthesis method similar to CEGIS [44] for designing protocols based on communicating variants of finite state machines. Here, the algorithm requires a temporal logic specification and a set of concolic snippets which describe the transition update behaviour. First, a synthesis engine consisting of an expression enumerator and an SMT solver, completes each transition such that the transition update corresponds to the given scenarios. Then, a model-checker checks the resulting implementation against the given specification. The design process is finished if the protocol implementation satisfies the specification, otherwise the user needs to

amend the implementation with respect to the counter-example provided by the model-checker, and then restart the synthesis tool.

Chapter 3

Preliminaries

3.1 Labelled Transition System

A *labelled transition system* (LTS) is a quadruple which consists of a state set T, an initial state $\text{init} \in T$, a transition function δ, and a state labelling function o. It takes inputs E from an external entity (i.e., the environment), and provides outputs O, where the assignment of O depends on the transition system's state.

$$\begin{aligned}&\mathcal{T} = (T, \text{init}, \delta, o)\\&\text{init} \in T\\&\delta : (T \times \mathbb{B}^E) \to T\\&o : T \to \mathbb{B}^O\end{aligned}$$

Note that $\mathbb{B}^X$ (also $\mathcal{P}(X)$) denotes the set of possible assignments to a set of variables X. The transition function δ determines the successor states given the current state and the current assignment of input variables. A *path* of an LTS is a possibly infinite sequence $(t_1, e_1)\,(t_2, e_2)\,(t_3, e_3)\ldots$ of pairs, where $t_i \in T$ denotes the state, and $e_i \in \mathbb{B}^E$ is the input valuation in the i-th step [49]. A *run* of an LTS is a maximal path starting in the initial state. A *trace* is a sequence $o(t_1)\,o(t_2)\ldots$ of output (also called state label) assignments corresponding to a given path.

An LTS is *closed* if there are no inputs. In this case, the transition function only depends on the current state. By contrast, an *open* LTS has a non-empty set of inputs. The output function assigns a value to each output variable. An LTS is *state identifiable* if the output function is such that each state is uniquely identifiable given an assignment of O.

$$\forall t, t' \in T : t \neq t' \to o(t) \neq o(t')$$

An *input-preserving* LTS is an LTS where the last read input value is part of the current state's label, i.e., $E \subseteq O$, and for each transition $t \xrightarrow{e} t'$, $o_E(t') = e$, where $o_E : T \to \mathbb{B}^E$ is the valuation of the output subset E in the particular state. Because an LTS enters the initial state before receiving any environment inputs, we use the empty symbol ϵ as the last read input value in the initial state, also called *initial direction*.

Non-Deterministic Labelled Transition System In a *non-deterministic* LTS, successor states are determined in a non-deterministic way. That is, the current state and current environment input are not mapped to a single successor state, but to a set of possible successor states. In order to model such a system, we define δ to be a relation.

$$\delta : (T \times \mathbb{B}^E) \times T$$

If a tuple (t, e, t') is in δ, there exists a transition $t \xrightarrow{e} t'$ (i.e., a transition from t to t' labelled with input e). In a non-deterministic closed LTS the transition relation can be defined as $\delta : T \times T$.

A *strongly connected component* (SCC) is a subset of the state set T, where each state is (not necessarily directly) reachable via δ from each other state.

3.2 Temporal Logics

Temporal logics extend propositional and first-order logic by operators which allow to specify terms that describe the evolution of a certain expression's truth value over time. Consider a set of atomic propositions AP. Each atom $p \in AP$ has one particular truth value in classical logic. By contrast, in temporal logic it has one truth value in each time step. We distinguish between temporal logics that are based on a linear-time perspective, and temporal logics that are based on a branching-time perspective [9]. We first focus on the branching time logic CTL*, and then introduce the more restricted logics CTL and LTL.

3.2.1 Computational Tree Logic*

Computational Tree Logic* (CTL*) is a branching time logic [29]. It distinguishes between *state formulae* and *path formulae*. A well-formed Computational Tree Logic* state formula over AP has the following syntax.

$$\Phi ::= \mathsf{true} \mid a \mid \Phi \wedge \Psi \mid \neg\Phi \mid \mathsf{E}\varphi$$

where $a \in AP$ is an atomic proposition, Φ and Ψ are state formulae, and φ is a path formula. Path formulae have the following syntax.

$$\varphi ::= \Phi \mid \varphi \wedge \psi \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\psi$$

where Φ is a state-formula, and φ, ψ are path formulae [9]. Moreover, we use the following derived operators.

$\varphi \vee \psi$ is equivalent to $\neg(\neg\varphi \wedge \neg\psi)$

$\varphi \implies \psi$ is equivalent to $\neg\varphi \vee \psi$

$\varphi \iff \psi$ is equivalent to $(\varphi \implies \psi) \wedge (\psi \implies \varphi)$

$\mathsf{F}\varphi$ is equivalent to $\mathsf{true}\mathsf{U}\varphi$

$\mathsf{G}\varphi$ is equivalent to $\neg\mathsf{F}\neg\varphi$

$\varphi\mathsf{W}\psi$ is equivalent to $(\varphi\mathsf{U}\psi) \vee \mathsf{G}\varphi$

$\mathsf{A}\varphi$ is equivalent to $\neg\,\mathsf{E}\,\neg\varphi$

The operators E and A are called *path quantifiers*. Let $\mathcal{T} = (T, \mathrm{Init}, \delta, o)$ be a transition system with state set T, initial states $\mathrm{Init} \subseteq T$, transition relation δ, and labelling function $o : T \to \mathbb{B}^{AP}$. We use π to denote a path $t_0\ t_1\ t_2 \ldots$ of the transition system, and π^i with $i \geq 0$ to denote a suffix $t_i\ t_{i+1} \ldots$ of π. Moreover, let $a \in AP$ be an atomic proposition, $t \in T$ a state, Φ and Ψ state formulae, and φ, ψ path formulae. Then the semantics the satisfaction relation $\models$ is defined as follows [9].

For state formulae,

$t \models \mathsf{true}$

$t \models a$ iff $a \in o(t)$

$t \models \Phi \wedge \Psi$ iff $t \models \Phi$ and $t \models \Psi$

$t \models \neg\Phi$ iff $t \not\models \Phi$

$t \models \mathsf{E}\varphi$ iff there exists a path π starting in t that satisfies $\pi \models \varphi$

$\mathcal{T} \models \Phi$ iff $\forall t \in \text{Init} : t \models \Phi$

For path formulae,

$\pi \models \Phi$ iff $t_0 \models \Phi$

$\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$

$\pi \models \neg\varphi$ iff $\pi \not\models \varphi$

$\pi \models \mathsf{X}\varphi$ iff $\pi^1 \models \varphi$

$\pi \models \phi\mathsf{U}\psi$ iff $\exists j \geq 0 : \pi^j \models \psi$ and $\forall\, 0 \leq i < j : \pi^i \models \varphi$

3.2.2 Linear-Time Temporal Logic

Linear-time temporal logic (LTL) was defined by Pnueli [64]. In contrast to branching time logics like CTL*, it provides only limited support for branching sequences. The syntax of LTL is defined as follows.

$$\varphi ::= \bot \mid \mathsf{true} \mid a \mid \varphi \wedge \psi \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\psi$$

where a is an element of the set of atomic propositions AP, and φ, ψ are valid LTL formulae. As for CTL*, we can introduce derived operators $\vee$, $\Longrightarrow$ etc. based on this syntax. Note that the LTL syntax does not allow the use of path quantifiers (E, A). However, every well-formed LTL formula is also a well-formed CTL* path formula. Furthermore, the semantics of an LTL formula φ is equivalent to the semantics of the CTL* state formula $\mathsf{A}\varphi$. Hence, LTL is a sublogic of CTL*. The logic LTL\X is equivalent to LTL without the X operator.

3.2.3 Computational Tree Logic

Computational Tree Logic (CTL) is a sublogic of CTL* [19]. Here, each path formula must be immediately preceded by a path quantifier (A, E). This restriction results in the following syntax for CTL.

$$\Phi ::= \mathsf{true} \mid a \mid \Phi \wedge \Psi \mid \neg\Phi \mid \mathsf{E}\Phi$$

$$\varphi ::= \mathsf{X}\Phi \mid \Phi\mathsf{U}\Psi$$

where $a \in AP$ is an atomic proposition, Φ and Ψ are state formulae, and φ is a path formula. The semantics is as defined for CTL*.

3.2.4 Linear-time Temporal Logic and Computational Tree Logic

Both CTL and LTL are sublogics of CTL*. The expressiveness of CTL and LTL is incomparable, i.e., each logic contains properties that cannot expressed in the other. In particular, the restriction that each path formula must be preceded by a path quantifier in CTL applies to a subset of LTL formulas which cannot be expressed in CTL. For example, consider the LTL formula $\mathsf{G}(r \rightarrow \mathsf{F}g)$, where r is the request signal and g is the grant signal of an arbiter. The equivalent CTL* formula is $\mathsf{AG}\,(r \rightarrow \mathsf{F}g)$, which is not supported by CTL. Valid CTL formulas like $\mathsf{AG}\,(r \rightarrow \mathsf{AF}g)$ or $\mathsf{AG}\,(r \rightarrow \mathsf{EF}g)$ do not have the same semantics. LTL and CTL* formulas express that for all paths, if there is a request r, a grant g must eventually follow in the same path, whereas the CTL formulas express that for all paths, if there is a request r, there must eventually be a grant g in all paths (some path, respectively). However, in CTL, we cannot force a grant in the path in which the request is received.

3.3 Satisfiability Modulo Theories

Many real-world problems can be described as constraint satisfaction problems [27]. One popular example is the propositional satisfiability (SAT) problem, which is to decide whether a propositional logical formula over Boolean variables is satisfiable. The satisfiability modulo theories (SMT) problem extends the classical NP-complete SAT problem by supporting theories and thus allows to formalize problems that require additional expressiveness. Examples for such *background theories* are arithmetics, arrays, and the theory of bit vectors. SMT solvers therefore unite the capabilities of a) SAT solvers, which aim at finding assignments for propositional variables that satisfy logical formulae, and b) theory solvers, which rely on decision procedures that exploit the mathematical properties of the particular theory in order to determine solutions for variables having a type defined by the theory (a so-called *sort*) .

Like SAT solvers, SMT solvers perform a case-analysis. To this end, most implementations rely on *systematic search*. Here, the search space consisting of all possible variable assignments is represented as a tree, where each vertex corresponds to the propositional variables, and the (two) outgoing edges of each vertex represent the variable assignments true and false, respectively. Each path from the root node to a leaf node represents an assignment to all variables. The goal of systematic search is then to systematically explore the tree and to find an appropriate path. The most widely used systematic search algorithm in the field of SAT solving is DPLL, which operates on CNF formulae [26].

Most well-performing SMT solvers basically combine Boolean reasoning (DPLL, in particular) with theory reasoning (provided by the particular theory solver) as illustrated in Algorithm 1. In this so-called *offline lazy* approach the given formula is first converted in to a propositional CNF formula (Line 1). To this end, variables and expressions belonging to a background theory are abstracted by fresh propositional variables. For example, the arithmetic expression $(a \geq 0)$ is replaced by a propositional variable x, and $x == (a \geq 0)$. Then, the DPLL algorithm determines whether there exists a satisfying assignment for the propositional formula F (Line 3). If there is no such assignment, there is also no solution such that φ is satisfied, thus the algorithm returns UNSAT. Otherwise, the resulting assignment M (model) is applied to the background theory (e.g., $x == \mathsf{true}$ is equivalent to $a \geq 0$). In Line 6, the theory solver tries to find solutions for sort variables under consideration of the newly derived constraints. If an assignment is found, the solver terminates by returning SAT. Otherwise, a further loop execution is done. In order to avoid that the DPLL algorithm reports the same propositional variable assignment in succeeding runs, the formula F is extended by a so-called *blocking clause*. [43]

Real-world SMT solvers extend this basic algorithm by a plethora of optimizations. For example, the theory solver is usually used for checking the theory-side consistency of partial propositional variable assignments (*online integration*). Moreover, theory deduction rules are used to prune the search space (*theory propagation*). A further challenge in SMT solving is the combination of multiple theories. Here, it depends on the particular theories how they can be combined and whether their combination is

Data: Propositional formula φ under theory T
Result: SAT or UNSAT
$F \leftarrow$ CNF_bool(φ)
while true **do**
  res, $M \leftarrow$ check_SAT(F)
  if *res* == true **then**
    $M' \leftarrow$ to_T(M)
    res $\leftarrow$ check_T(M')
    if *res* == true **then**
      return SAT
    else
      $F \leftarrow F \cup$ blocking_clause(M)
    end
  else
    return UNSAT
  end
end

Algorithm 1: Offline lazy SMT solving approach [43]

decidable. *Strongly disjoint* theories, which do not share common data types (and consequently have no common variables) can be easily combined by solving each part of the formula under consideration of the particular background theory. Theories which can be combined that way are e.g., the theory of arithmetics and the theory of bit vectors. Theories that do not have any common predicate and function symbols, but possibly have common sort symbols (*disjoint* theories) can be combined using the Nelson-Oppen procedure [62] if they are stably infinite, i.e., every formula satisfiable in the theory is also satisfiable in an infinite model of the theory. Extensions of the Nelson-Oppen method enable the combination of non-stably infinite theories, and non-disjoint theories [26].

Most theories are only decidable in their quantifier-free fragment. Yet defining an SMT problem often involves the use of quantifiers. For this reason, many SMT solvers implement an incomplete approach to support quantifiers using existential quantifier elimination and heuristics-based instantiation of universal quantifiers [50].

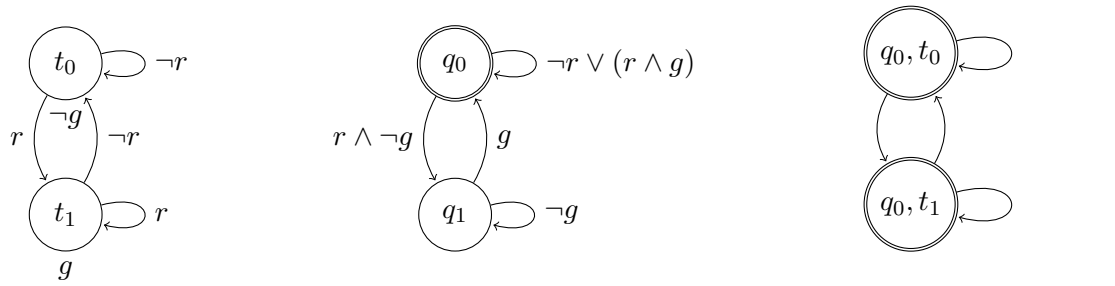
3.4 Tree Automaton

An *alternating tree automaton* $\mathcal{U}$ is a tuple $(\Sigma, \Upsilon, Q, q_0, \Delta, \Phi_{\text{acc}})$ where Σ is a finite set of labels, Υ is a finite set of directions, Q is the finite set of states, q_0 is the initial state, and Φ_{acc} is an acceptance condition [38]. The transition function $\Delta : (Q \times \Sigma) \rightarrow \mathbb{B}^+(\Upsilon \times Q)$ maps each state-label-pair $(q, \sigma) \in Q \times \Sigma$ to a positive Boolean combination of state-direction pairs. In a *non-deterministic* alternating tree automaton the disjunctive forms of formulas in the image set of Δ are such that each disjunct contains at most one element of $Q \times \{v\}$ for each $v \in \Upsilon$. An alternating tree automaton where the image of Δ is restricted to conjunctions of elements of $Q \times \Upsilon$ is called *universal*. A universal and non-deterministic tree automaton is *deterministic* [38].

A run of an alternating tree automaton $\mathcal{U}$ on an LTS $\mathcal{T}$ results in a *run graph* $\mathcal{G} = (G, E)$ with nodes G ($G \subseteq Q \times T$) and edges E. Note that the directions of the automaton's Υ set serve as inputs for the LTS, while the LTS outputs correspond to the automaton's current label values. A run graph has the following properties. First, the initial state (q_0, t_0) must be in G. Second, for each state (q, t), all elements of the set $\{(q', v) \in Q \times \Upsilon \mid ((q, t), (q', \delta(t, v)) \in E\}$ must satisfy the formula $\Delta(q, o(t))$ [38]. Thus, there must exist a corresponding transition in both the tree automaton and the LTS for each edge $(q, t) \rightarrow (q', t')$. To be precise, the current LTS outputs need to be such that $\Delta(q, o(t))$ is true, and given

the automaton's label v, t' is a successor of t regarding the LTS transition function δ.

A *path* of the run graph is a sequence of graph vertices $g_1\, g_2\, g_3 \ldots$, where $g_1 = (q_0, \text{init})$ and each vertex in the sequence has a single successor. A run graph is *accepting* if every infinite path satisfies the alternating tree automaton's acceptance condition. There exist several acceptance conditions. For example, the alternating parity tree automaton defines a coloring function $\alpha : Q \to C \subset \mathbb{N}$. A path $g_1\, g_2\, g_3 \ldots$ satisfies the parity condition if the maximum of all infinitely often occurring values in the corresponding coloring function sequence $\alpha(g_1)\, \alpha(g_2)\, \alpha(g_3) \ldots$ is even. The overall run graph is accepted if all paths satisfy the parity condition. Alternating Büchi tree automata define a set $F \subseteq Q$ of accepted states. The acceptance condition is satisfied if all infinitely often visited states are in F. By contrast, the co-Büchi acceptance condition is satisfied if states in F are visited only finitely often. Both the Büchi and the co-Büchi acceptance condition can be expressed by the parity condition. Here, the coloring function's image is restricted to a set with two elements ($\{1, 2\}$ for the Büchi condition, $\{0, 1\}$ for the co-Büchi condition), and elements in F are mapped to the respective higher number (2 for the Büchi condition, 1 for the co-Büchi condition) [38].



(a) Example LTS

(b) Büchi Tree Automaton for LTL formula $\mathsf{G}(r \to \mathsf{F}\, g)$

(c) Run graph for UCT on LTS

Figure 3.1: Büchi Automaton and its run on a small LTS

For example, consider the LTS shown in Figure 3.1a. It has two states $T = \{t_0, t_1\}$ with initial state t_0, an input signal r (request), and an output signal g (grant). The Büchi tree automaton (BTA) in Figure 3.1b accepts all runs that satisfy the the LTL formula $\mathsf{G}(r \to \mathsf{F}\, g)$. This automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \Delta, F)$ has labels $\Sigma = \mathbb{B}^E \times \mathbb{B}$, directions $\Upsilon = \mathbb{B}^E$, $Q = \{q_0, q_1\}$ with initial state q_0, transition function Δ, and the set of accepting states $F = \{q_0\}$.

The run graph for the BTA on the given LTS is shown in Figure 3.1c. The node (q_0, t_0) corresponds to the initial state of both the BTA and the LTS. Whenever there is a request (r raised), the LTS immediately moves to state t_1 (granting state), and thus the BTA remains in q_0. The transition from t_1 back to t_0 also satisfies the UCT's loop edge on q_0, therefore the UCT remains in q_0. The same applies to the loop transitions for t_0 and t_1. For this reason, the rejecting UCT state q_1 is not present in run graph node and the LTS does not contain a rejecting run.

3.5 Distributed System

3.5.1 Architecture

Jacobs and Bloem [49] define an architecture as a tuple $A = (P, env, V, E, O)$, where P is a finite set of processes that contains the environment process and system processes $P^- = P \setminus \{env\}$. V is the set of so-called *system variables*, i.e., outputs of all processes in P. $E = \{E_i \subseteq V | i \in P^-\}$ contains a set of input variables for each system process, and $O = \{O_i \subseteq V | i \in P\}$ defines the outputs of each process such that $V = \bigcup_{i \in P} O_i$. System variables have exactly one origin (each is present in O_i for a single

$i \in P$), but can appear in an arbitrary number of input sets E_i. In a *fully informed* architecture, $E_i = E_j$ for all $i, j \in P^-$. A set of system processes with $|P^-| \geq 1$ is called *distributed system*.

3.5.2 Implementation

Given an architecture A, the *implementation* of a system process $i \in P^-$ with inputs E_i and outputs O_i is an LTS $\mathcal{T}_i = (T_i, \text{init}_i, \delta_i, o_i)$, with $\delta_i : T_i \times \mathbb{B}^{E_i} \times T_i$ and $o_i : T_i \to \mathbb{B}^{O_i}$ [49]. Let $\{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ be the set of system process implementations, where $\mathcal{T}_i$ is an implementation for a particular $P_i \in P^-$. Then, the *composition* of all system process implementations is an LTS $\mathcal{S} = (S, \text{init}_S, \delta_S, o_S)$ with inputs E_S, schedulings Sched_S, and outputs O_S, where

- $E_S = (E_1, \ldots, E_n)$
- $\text{Sched}_S = \underbrace{\mathbb{B} \times \cdots \times \mathbb{B}}_{n \text{ times}}$
- $O_S = \bigcup_{i \in P^-} O_i$
- $S = T_1 \times \cdots \times T_n$
- $\text{init}_S = (\text{init}_1, \ldots, \text{init}_n)$,
- $o_S : S \to \mathbb{B}^{O_S}$
  $o_S(t_1, \ldots, t_n) = o_1(t_1) \cup \cdots \cup o_n(t_n)$

A state $s \in S$ of the composition is called *global state*. The scheduling signals $\text{sched} = (\text{sched}_1, \ldots, \text{sched}_n)$ are provided by a scheduler, which is part of the environment and decides for each process whether it is allowed to move. Let δ_i for each system process implementation $\mathcal{T}_i$ be a transition function (that is, $\delta_i : T_i \times \mathbb{B}^{E_i} \to T_i$). Then, $\delta_S : S \times E_S \times \text{Sched}_S \to S$ is as follows.

$$\delta_S((t_1, \ldots, t_n), (e_1, \ldots, e_n), (\text{sched}_1, \ldots, \text{sched}_n)) = (\delta'_1(t_1, e_1, \text{sched}_1), \ldots, \delta'_n(t_n, e_n, \text{sched}_n))$$

$$\delta'_i(t_i, e_i, \text{sched}_i) = \begin{cases} \delta(t_i, e_i) & \text{if sched}_i \\ t_i & \text{otherwise} \end{cases}$$

For compositions of system process implementations $\mathcal{T}_i$ with transition relations $\delta_i : (T_i \times \mathbb{B}^{E_i}) \times T_i$, $\delta_S : (S \times \mathbb{B}^{E_S} \times \text{Sched}_S) \times S$ is as follows.

$$((t_1, \ldots, t_n), (e_1, \ldots, e_n), (\text{sched}_1, \ldots, \text{sched}_n), (t'_1, \ldots, t'_n)) \in \delta_S \Leftrightarrow$$
$$\big(\text{sched}_1 \wedge (t_1, e_1, t'_1) \in \delta_1 \vee \neg\text{sched}_1 \wedge t'_1 = t_1\big) \wedge \cdots \wedge \big(\text{sched}_n \wedge (t_n, e_n, t'_n) \in \delta_n \vee \neg\text{sched}_n \wedge t'_n = t_n\big)$$

Each component $t_i \in T_i$ of s corresponding to a certain system process implementation is a so-called *local state*, also written as $s(i)$. Given a global input $e \in E_S$, the local input for process i is denoted as $e(i)$. A *configuration* of a system is a tuple $(s, e, \text{sched}) \in S \times \mathbb{B}^{E_S} \times \text{Sched}_S$.

3.5.3 Paths and Runs

A *path* of a composition (also called *global path*) is a possibly infinite sequence of configurations $(s_0, e_0, \text{sched}_0)\,(s_1, e_1, \text{sched}_1)\ldots$ where $s_{i+1} = \delta_S(s_i, e_i, \text{sched}_i)$, and $(s_i, e_i, \text{sched}_i, s_{i+1}) \in \delta_S$, respectively. Given a global path $x = (s_i, e_i, \text{sched}_i)\,(s_{i+1}, e_{i+1}, \text{sched}_{i+1})\ldots$, a *local* path $x(p) = (s_i(p), e_i(p))\,(s_{i+1}(p), e_{i+1}(p))\ldots$ is its projection to a single process p. A *run* of a composition is a maximal path starting in the global initial state.

A *synchronous* system is a composition which only allows configurations where all processes are scheduled. Considering such systems, the configuration's Sched_S part can be omitted. By contrast, an

asynchronous system is a composition that only allows configurations where a single process is scheduled. The configuration of an asynchronous system can also be defined as a triple (s, e, p), where s and e are as described above, and $p \in P^-$ is not a tuple of Boolean variables, but contains the scheduled process. Consequently, a path of an asynchronous system is a sequence $(s_0, e_0, p_0)\,(s_1, e_1, p_1)\ldots$, where $\forall p' \in P^- : (p' \neq p_i) \rightarrow (s_{i+1}(p') = s_i(p'))$, and there is a transition $s_i(p_i) \xrightarrow{e_i(p_i)} s_{i+1}(p_i)$. Likewise, a run of an asynchronous system is defined as above for the general case.

A scheduling is *fair* if a certain *fairness condition* is satisfied. Furthermore, a *fair run* is a run that is based on a fair scheduling. An example for a fairness condition is the requirement that each process is scheduled infinitely often. In Section 6.2.3 we describe and analyze appropriate fairness conditions under consideration of our system model.

3.6 Model Checking and Synthesis

Given an architecture A, a *specification* φ is a logic formula over system variables V of A that represents the desired behaviour of a distributed system. Specifications are usually conjunctions of so-called *properties*. Each property is an implication

$$\text{assumption} \rightarrow \text{guarantee}$$

where the left-hand side describes preconditions, e.g., environment behaviour etc., and the right-hand side is the required system behaviour in case all preconditions are fulfilled. If the left-hand side of a property is true, the particular guarantee must hold unconditionally. This is equivalent to a property that only consists of the guarantee part.

We distinguish between two types of properties. A *safety property* describes that some "bad thing" must never happen, while a *liveness property* describes that some "good thing" will eventually happen [59, 1].

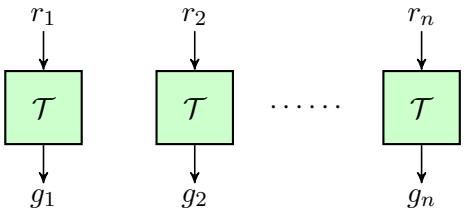


Figure 3.2: Arbiter processes

For example, consider a set of arbiter processes, as shown in Figure 3.2, where each process gets a request signal r_i from the environment, and provides a grant signal g_i to the environment. Let the specification for a system of two processes be as follows.

$$\mathsf{G}(r_1 \rightarrow \mathsf{F}(g_1)) \wedge \mathsf{G}(r_2 \rightarrow \mathsf{F}(g_2)) \wedge \mathsf{G}(\neg(g_1 \wedge g_2))$$

The specification consists of three properties. All properties are without precondition (assumption), thus the guarantees have to hold unconditionally. The first two conjuncts are liveness properties. They stipulate that each request must be finally granted (something "good" must eventually happen). By contrast, the third property is a safety property, since it describes the absence of something "bad" (i.e., mutual exclusion).

A synchronous distributed system *satisfies* the specification φ if for all runs $(s_0, e_0)\ldots$ the set $(o(s_j) \cup e_j)$ satisfies φ in each step j. A formula φ is satisfied by an asynchronous system if for all runs $(s_0, e_0, p_0)\ (s_1, e_1, p_1)\ldots$, the set $(o(s_j) \cup e_j)$ satisfies φ in each step j. Note that for asynchronous systems, some (liveness) properties are possibly only realizable if the scheduling is fair. For such properties, the assumptions part needs to contain an appropriate fairness constraint such that the system only needs to satisfy the guarantee if the fairness condition holds.

3.6.1 Model Checking and Synthesis

The *model checking problem* is to decide whether the composition of a given set of implementations for system processes P^- satisfies the specification φ, that is, $(\mathcal{T}_1, \ldots, \mathcal{T}_n) \models \varphi$. [7]

Given an architecture A and a specification φ, the *distributed synthesis problem* is to decide whether there exists an implementation $\mathcal{T}_i$ for each $i \in P^-$ such that the specification is satisfied by the composition $\mathcal{S}$ of these processes, i.e., $A, \mathcal{S} \models \varphi$. If there exists such an implementation, the specification is *realizable* [49]. Pnueli and Rosner [66] showed that the distributed synthesis problem is undecidable for the case where system processes are not fully informed. The synthesis of distributed systems based on fully informed architectures is decidable, but the complexity of this problem is nonelementary [37]. Moreover, the synthesis problem for asynchronous systems has been shown to be undecidable [68].

3.6.2 Bounded Synthesis

The *bounded synthesis problem* is a specialization of the general synthesis problem stated in Section 3.6.1. Here, the goal is to find implementations with a given maximal size (i.e., number of LTS states) that satisfy a certain LTL specification φ. More precisely, given an architecture A, a specification φ, a set of bounds $b = \{b_1, b_2, \ldots, b_n\}$, and a bound b_S, the goal is to find an implementation $\mathcal{T}_i$ for each system process $i \in P^-$ such that φ is satisfied, $|T_i| \leq b_i$, and for the composition $\mathcal{S}$, $|S| \leq b_S$. [38, 49] Finkbeiner and Schewe [69, 38] showed that bounded synthesis is semi-decidable, and introduced an algorithm to solve the bounded synthesis problem. This algorithm consists of the following main steps:

1. Fix the maximum size of system process implementations by setting bounds b and b_S
2. Construct a universal co-Büchi tree automaton (UCT) which accepts all runs that violate φ.
3. Solve the problem of finding process implementations such that there is no run in the composition $\mathcal{S}$ that is accepted by the UCT. If there is no such system, increase the bound values, and continue with the first step.

In the second step, the LTL formula is converted into a UCT as described by Kupferman and Vardi [57]. Here, the LTL specification φ is negated, and then converted into a nondeterministic Büchi word automaton (NBW). As a final step of the conversion, a UCT that simulates the NBW is constructed. The algorithm yields a single exponential blow-up caused by the construction of the NBW that accepts φ.

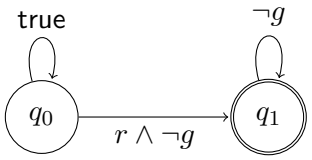


Figure 3.3: Universal co-Büchi Tree Automaton for the property $\mathsf{G}(r_1 \rightarrow \mathsf{F}(g_1))$

The resulting tree automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, q_0, \Delta, F)$ accepts all "bad" runs, i.e., runs that violate property φ. For example, consider Figure 3.3, which corresponds to the arbiter liveness property $\mathsf{G}(r_1 \rightarrow \mathsf{F}(g_1))$, where r_1 is the request signal provided by the environment to the first system process and g_1 is the grant signal provided by the first system process. As defined by the co-Büchi condition, a path is "bad", if a rejecting state is visited infinitely often. For our example, a bad path is obviously a path where a request is raised, but not granted. Let $\mathcal{G} = (G, E)$ be the run graph of $\mathcal{U}$ on the desired system composition $\mathcal{S}$ (in our example a single LTS). If the system composition violates the specification, there is a path in the run graph such that the sequence of Q components contains some rejecting state infinitely often.

We observe that finding a valid implementation is equivalent to asking for a run graph that contains only paths with finitely many rejecting UCT states. Finkbeiner and Schewe [38] solve this problem as a labelling problem (which is encoded and solved as an SMT instance). They introduce an annotation function $\lambda : Q \times S \to \mathbb{N} \cup \{\_\}$ that assigns either a natural number or an empty symbol to each vertex (q, s) of the run graph. The annotation function must satisfy the following constraints:

- $\lambda(q_0, \text{init}_S) = 0$
- Each state $(q', s') \in G$ that is reachable from a state (q, s) with $\lambda(q, s) \neq \_$ must be annotated such that $\lambda(q', s') \triangleright \lambda(q, s)$, where $\triangleright \in \{>, \geq\}$, and $>$ iff $q' \in F$.

If there exists a labelling that satisfies the above conditions, there exists a valid implementation. In [38], the authors describe how the annotation problem is encoded in SMT. States are represented by natural numbers $S = \{1, \ldots, b_S\}$. The transition function is represented as an uninterpreted function $\delta_S : S \times \mathbb{B}^{E_S} \to S$.[1] For each output $o \in O_S$, a function $o : S \to \mathbb{B}$ is introduced.

Because we not only want to find the composition $\mathcal{S}$, but the actual implementations $\mathcal{T}_1, \ldots, \mathcal{T}_n$, we add for each system process i a projection $d_i : S \to T_i$ that maps each global state to a local state of the particular process implementation. Here, $T_i = \{1, \ldots, b_i\}$. Moreover, we need to ensure that each local transition δ_i only depends on inputs that the particular process gets from the environment. Likewise, each process' output must only depend on the particular local state. In case of asynchronous systems, we need additional constraints to ensure that for each global transition $s \to s'$, s and s' only differ in the scheduled process' local state.

3.6.3 Parameterized Synthesis

A *parameterized architecture* is a function $\Pi : \mathbb{N} \to \mathcal{A}$ that selects an architecture $A \in \mathcal{A}$ for a given number of processes $n \in N$. [49] A *parameterized specification* Φ is a formula in temporal logic (LTL, CTL, CTL*) over indexed variables that are universally or existentially quantified in prenex form [49]. The *instantiation* of a parameterized specification for $n \in \mathbb{N}$ processes, written as $\Phi(n)$, is a specification φ that is obtained by replacing all process-quantified formulae by their instantiations.

- $\forall i : h(i)$ is instantiated to $\bigwedge_{i \in [1,n]} h(i)$
- $\exists i : h(i)$ is instantiated to $\bigvee_{i \in [1,n]} h(i)$

Parameterized architectures require that architecture instantiations in the class $\mathcal{A}$ are *similar*, that is, the implementation of system processes are either isomorphic or belong to a group of implementations. Consider a parameterized architecture with isomorphic processes, i.e., $\forall A \in \mathcal{A} \forall i, j \in P^- : \mathcal{T}_i = \mathcal{T}_j$, where $\mathcal{T}_i$ denotes the implementation of system process $i \in P^-$. In other words, there is a single implementation $\mathcal{T}$ for all system processes. Then, a parameterized architecture Π and a process implementation *satisfy* a parameterized specification Φ if for all $n \in N$, $\Pi(n), (\underbrace{\mathcal{T}, \ldots, \mathcal{T}}_{n \text{ times}}) \models \Phi(n)$. This is written as $\Pi, \mathcal{T} \models \Phi$ [49]. Parameterized architectures with multiple groups of isomorphic processes are defined in a similar way (see Chapter 4).

The *parameterized model checking problem* (PMCP) is to decide whether $\Pi, \mathcal{T} \models \Phi$ holds for a given parameterized architecture Π, a parameterized specification Φ, and a system process implementation $\mathcal{T}$. The *parameterized synthesis problem* for isomorphic processes is to find an implementation $\mathcal{T}$ such that $\Pi, \mathcal{T} \models \Phi$ [49].

[1] A transition relation can likewise be represented as an uninterpreted function $\delta_S : S \times \mathbb{B}^{E_S} \times S \to \mathbb{B}$ that is `true` for some tuples $(s, e, s') \in (S \times \mathbb{B}^{E_S} \times S)$ iff there is a transition from s to successor state s' with inputs e.

Chapter 4

System Model

In this section, we first introduce the class of guarded systems, and then define the parameterized model checking and synthesis problem for this system class.

4.1 Guarded System

4.1.1 Process Template and Guarded Protocol

A *guarded system* is an asynchronous distributed system, where each system process is based on a so-called process template. Given a set of environment inputs E_k and a set of outputs O_k, a *process template* is a special form of LTS $\mathcal{T}_k = (T_k, \text{init}_k, \delta_k, \text{guard}_k, o_k)$, where T_k denotes the set of states, $\text{init}_k \in T_k$ is the initial state, $\delta_k : T_k \times \mathbb{B}^{E_k} \times T_k$ is the transition relation, and $o_k : T_k \to \mathbb{B}^{O_k}$ is the output (state labelling) function, as for standard LTSs defined in Chapter 3. Let the tuple $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\text{k})$ with *template index* $k \in [1, \text{k}]$ define the process templates of a system, also called *system type*. Then, the *guard assignment function* $\text{guard}_k : T_k \times \mathbb{B}^{E_k} \times T_k \to \mathbb{B}^G$ with $G = \bigcup_{k \in [1,\text{k}]} \mathbb{B}^{O_k}$ maps each transition in δ_k to a so-called *guard*. Every element of $\mathbb{B}^G$ can be seen as a subset of G. We call such a subset a *guard set*. Elements contained in a guard set are denoted as *guard variables*. Because each guard variable represents one assignment of output variables for a particular process template, we also say that a guard variable represents an "observable state" [7]. All processes that are based on the same process template form a group of isomorphic processes. In a guarded system, there is a finite number of such process groups.

The *guarded protocol* $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\text{k})^{(n_1, n_2, \ldots, n_\text{k})}$ is the interleaving parallel composition of n_1 instances of the first process template, n_2 instances of the second process template, and so forth [7]. We call the tuple $(n_1, \ldots, n_\text{k})$ *multiplicity vector*. The set of the n_k instances that are based on template $\mathcal{T}_k$ are also denoted as $\mathcal{T}_k$-processes. By $\mathcal{T}_k^i$ we denote the i-th instance of process template $\mathcal{T}_k$. The global state of a guarded protocol $(\mathcal{T}_1, \ldots, \mathcal{T}_\text{k})^{(n_1, \ldots, n_\text{k})}$ is a tuple $s = (t_1^1, \ldots, t_1^{n_1}, t_2^1, \ldots, t_\text{k}^{n_\text{k}})$, and $s(\mathcal{T}_k^i) = t_k^i$ denotes the local state of process $\mathcal{T}_k^i$. $s_j(\mathcal{T}_k^i)$ is the local state of process $\mathcal{T}_k^i$ in the j-th step. Likewise, given the environment input e, $e(\mathcal{T}_k^i)$ denotes the local input for process $\mathcal{T}_k^i$, and $e_j(\mathcal{T}_k^i)$ the local input for process $\mathcal{T}_k^i$ in the j-th step. By $s(\mathcal{T}_k)$ (also abbreviated to $s(k)$) we denote the tuple that contains the local states of all instances based on process template $\mathcal{T}_k$ (also called $\mathcal{T}_k$-states). $e(\mathcal{T}_k)$ denotes the inputs for these processes.

4.1.2 Guard Evaluation

Consider a guarded protocol with k process templates $\{\mathcal{T}_1, \ldots, \mathcal{T}_\text{k}\}$, and let $N = \sum_{k \in [1,\text{k}]} |\mathbb{B}^{O_k}|$. Given a global state $s = (t_1^1, \ldots, t_1^{n_1}, t_2^1, \ldots, t_\text{k}^{n_\text{k}})$, a local transition $t \xrightarrow{e_\text{trans}} t'$ of a process $\mathcal{T}_k^i$ is *enabled* if

the following conditions are satisfied: a) the environment inputs $e(\mathcal{T}_k^i)$ match e_{trans}, and b) the guard resulting from $\text{guard}_k(t, e_{\text{trans}}, t')$ is satisfied, as defined in the following.

Let state : $G \to \bigcup_k T_k$ be a bijective mapping from guard variable to a set of process template states for which the particular guard variable is true. The *state guard function* maps the current global state to a *global state guard*, and $g \in G$ is true in the global state guard for s iff there exists a process $p \neq \mathcal{T}_k^i$ (i.e., the process whose transition is considered) such that $s(p) \in \text{state}(g)$.

The *guard evaluation function* eval_guard : $\mathbb{B}^G \times \mathbb{B}^G \to \mathbb{B}$ takes a global state guard and a transition guard, and evaluates to true iff the transition guard is *satisfied* by the current state guard. We distinguish between two kinds of guard evaluation functions, namely disjunctive guards and conjunctive guards. In a *disjunctive guarded* system, the transition guard is satisfied if there is at least one guard variable which is contained in both the desired transition's guard and the current global state guard. In a *conjunctive guarded* system, the transition guard is satisfied if all guard variables in the current global state guard are contained in the transition guard. Moreover, the transition guard must contain at least one guard variable. We define an additional restriction for conjunctive guards: Processes that are in their respective initial states do not cause the disabling of any transition [35]. To this end, each transition guard set must contain all guard variables that correspond to the process templates' initial states. A transition is *unguarded* if the system never reaches a global state that causes the guard evaluation function to evaluate to false for the transition's guard. Such transitions are equivalent to the transitions of standard LTSs.

In a guarded protocol where each template's output function is injective, we can omit the concept of guard sets and define the guard evaluation using state sets. That is, the guard assignment function guard_k is a mapping to $\mathbb{B}^{T_k}$, and the guard is evaluated by directly comparing it with the global state set (excluding the particular process' state). Furthermore, we use the term *1-guard* to denote a special guard set. In a disjunctive system, a 1-guard is a guard set with cardinality 1. In a conjunctive guarded system, a *1-guard* contains $N - 1$ guard variables, where N denotes the overall number of guard variables in the system.

4.1.3 Path and Run

Configuration, path and run are defined as for asynchronous systems. A guarded protocol *configuration* is a triple (s, e, p) with global state s, environment input e, and scheduled process p (or $\perp$ if there is no scheduled process). A *path* is a possibly infinite sequence of configurations $(s_0, e_0, p_0)\ (s_1, e_1, p_1) \ldots$, where s_{j+1} is the result of a local transition from s_j of process p_j. In each step the following conditions must be satisfied: a) the scheduled process p_j takes the enabled transition $s_j(p_j) \xrightarrow{e_j(p_j)} s_{j+1}(p_j)$, and b) for all other processes $p \neq p_j$, $s_{j+1}(p) = s_j(p)$. These requirements imply that the environment must schedule a process which is enabled with respect to the given inputs and the global state. $p_j = \perp$ iff there is no process with at least one enabled transition in step j. For paths of guarded protocols, the environment must not change the inputs of unscheduled processes, i.e., for all p, $e_{j+1}(p) = e_j(p)$, unless $p = p_j$. This definition is similar to the definition of an action-based system.

A *run* of a system is a maximal path starting in the initial state [7]. A path x *stutters* in step j if $x_{j-1} = x_j$. The *destuttering* of x, $\text{destutter}(x)$, is obtained by removing all stuttering steps from x. Two paths x and y are *stutter-equivalent* ($x \simeq y$) if $\text{destutter}(x) = \text{destutter}(y)$. Two systems are stutter-equivalent if for each infinite run of the first system, there is a corresponding stutter-equivalent infinite run in the second, and vice versa [7]. Stutter-equivalent paths and systems satisfy the same formulas in LTL\X, i.e., they are indistinguishable by any LTL\X formula. A run is *locally deadlocked* for some process p if there is some step j such that p is disabled for all $s_{j'}, e_{j'}$ with $j' \geq j$. A run is *globally deadlocked* if it is locally deadlocked for each process [7].

4.2 Restrictions

A guarded system requires that all guards are evaluated using the same guard evaluation function. In other words, it is not possible to mix conjunctive and disjunctive guards. We do not consider modifying the system model in order to support mixtures of guard evaluation functions, because the parameterized model checking problem was shown to be undecidable for such systems [30].

A further restriction is that the system model allows to define at most one transition for each tuple (t, e, t'). Consequently, transition systems as shown in Figure 4.1a are not directly supported. This problem does not apply to disjunctive guards. Given such an unsupported system (Figure 4.1a), we construct an equivalent supported one by transition replacement. To this end, we merge the two guard sets such that $\text{guard}_k(t, e, t') = g$, and $g = g_1 \cup g_2$. In other words, we build a disjunction of guard variables contained in the two guard sets (which are themselves interpreted as disjunctions by the disjunctive guard evaluation function). The resulting equivalent LTS for our example is shown in Figure 4.1b.

(a) Guarded template with unsupported transitions

(b) Equivalent solution (disjunctive guards)

Figure 4.1: The system model does not support multiple transitions with the same source state, destination state, and input label.

This merging is not applicable to conjunctive guarded systems. Let g be a merged transition guard. If a certain state guard satisfies g_1 (g_2), it also satisfies g. However, the opposite direction is not necessarily true. For example, consider the case where guard g is satisfied, and one subset of the global state guard satisfies g_1, but not g_2, whereas another subset of the global state guard satisfies g_2, but not g_1. Under the assumption that the environment provides inputs e, the merged transition (t, e, t') guarded with g is enabled, although both of the two original transitions are disabled.

A solution to this problem is to redefine the system model in order to support multiple transitions for the same triple (t, e, t'). That is, the process template function δ_k is modified such that it also includes the guard information.

$$\delta_k : T_k \times \mathbb{B}^{E_k} \times \mathbb{B}^G \times T_k$$

The guard assignment function guard_k therefore becomes superfluous. However, we forego this solution in our work, since it significantly increases the problem size of the synthesis instance, and was not shown to be indispensable in the course of our evaluation.

4.3 Model Checking and Synthesis

The *model checking problem* for guarded systems is to decide whether a given protocol $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})}$ satisfies a specification φ, i.e., $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})} \models \varphi$. The *(local) deadlock detection problem* is to decide whether for a system $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})}$ there exists a (locally) deadlocked run [7]. Given a multiplicity vector $(n_1, \ldots, n_\mathrm{k})$ and a specification φ, the *template synthesis problem* is to find process templates $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})$ such that $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})} \models \varphi$. Given a multiplicity vector $(n_1, \ldots, n_\mathrm{k})$, a tuple of bounds $(b_1, \ldots, b_\mathrm{k}) \in \mathbb{N} \times \cdots \times \mathbb{N}$, and a specification φ, the *bounded template synthesis problem* (also denoted as *bounded synthesis problem for guarded systems*) is to find templates $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})$ such that $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})} \models \varphi$ and $\forall k \in [1, \mathrm{k}] : |T_k| \leq b_k$. [7]

For guarded systems, we consider specifications Φ with the following structures, where $\Xi \in \{\mathsf{A}, \mathsf{E}\}$, and $h(\mathcal{T}_k^{i_k}, \ldots, \mathcal{T}_l^{i_l})$ is an LTL\X formula over in- and outputs of processes $\mathcal{T}_k^{i_k}, \ldots, \mathcal{T}_l^{i_l}$.

- Single-indexed [35]: $\bigwedge_{i\in[1,n_k]} \Xi h(\mathcal{T}_k^i)$
- Multi-indexed over one process template [7]: $\bigwedge_{i_1,\dots i_m\in[1,n_k]:i_1\neq\dots\neq i_m} \Xi h(\mathcal{T}_k^{i_1},\dots,\mathcal{T}_k^{i_m})$
- Multi-indexed over two process templates [7]: $\bigwedge_{i\in[1,n_k]} \bigwedge_{j_1,\dots j_m\in[1,n_l]:j_1\neq\dots\neq j_m} \Xi h(\mathcal{T}_k^i,\mathcal{T}_l^{j_1},\dots,\mathcal{T}_l^{j_m})$

The *parameterized model checking problem* for guarded systems is to decide whether given system type $(\mathcal{T}_1,\dots,\mathcal{T}_\mathrm{k})$, a parameterized specification Φ, and a multiplicity vector $(m_1,\dots,m_\mathrm{k})$, $(\mathcal{T}_1,\dots,\mathcal{T}_\mathrm{k})^{(n_1,\dots,n_\mathrm{k})} \models \Phi$ for all $(n_1,\dots,n_\mathrm{k}) \succeq (m_1,\dots,m_\mathrm{k})$.[1] The *parameterized template synthesis problem* is to find process templates $(\mathcal{T}_1,\dots,\mathcal{T}_\mathrm{k})$ given a multiplicity vector $(m_1,\dots,m_\mathrm{k})$ and a specification Φ such that $(\mathcal{T}_1,\dots,\mathcal{T}_\mathrm{k})^{(n_1,\dots,n_\mathrm{k})} \models \Phi$ for all $(n_1,\dots,n_\mathrm{k}) \succeq (m_1,\dots,m_\mathrm{k})$. [7]

[1] We use the symbol $\succeq$ for the component-wise comparison of two tuples: $(n_1,\dots,n_n) \succeq (m_1,\dots,m_n) \Leftrightarrow \forall i : n_i \geq m_i$

Chapter 5

Parameterized Model Checking for Guarded Systems

In this section we recapitulate already existing cutoff results for reducing the Parameterized Model Checking Problem (PMCP) for guarded systems to a standard Model Checking Problem of a system with a constant number of processes [35].

Consider the process templates $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})$ of a guarded system, and a supported parameterized specification Φ (as described in Chapter 4). Then, a *cutoff* is a multiplicity $(c_1, \ldots, c_\mathrm{k})$ such that

$$(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(c_1,\ldots,c_\mathrm{k})} \models \Phi \text{ iff } \forall (n_1, \ldots, n_\mathrm{k}) \succeq (c_1, \ldots, c_\mathrm{k}) : (\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})} \models \Phi$$

The system $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(c_1,\ldots,c_\mathrm{k})}$ is called *cutoff system*. The cutoff system satisfies the specification Φ if and only if this specification is also satisfied by any larger system. In order to check whether Φ holds for systems of any size, we have to check if $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(d_1,\ldots,d_\mathrm{k})} \models \Phi$ for all $(d_1, \ldots, d_\mathrm{k}) \preceq (c_1, \ldots, c_\mathrm{k})$. The cutoffs proved by Emerson and Kahlon depend on the number of template states and the specification's structure. For notational simplicity, the authors first consider a system type with two process templates $(\mathcal{T}_1, \mathcal{T}_2)$, and then generalize the resulting theorems to an arbitrary number of templates. We borrow this simplification for our explanations. Moreover, we use the notation (A, B) equivalently to $(\mathcal{T}_1, \mathcal{T}_2)$. By $\mathcal{T}_k$-process, we denote some process $\mathcal{T}_k^i$ based on template $\mathcal{T}_k$.

5.1 Disjunctive Cutoffs

Lemma 5.1.1 (Disjunctive Monotonicity Lemma [35])
For all closed disjunctive process templates (A, B)*,*

$$\forall n \geq 1 : (A, B)^{(1,n)} \models \mathsf{E}h(A^1) \Rightarrow (A, B)^{(1,n+1)} \models \mathsf{E}h(A^1)$$
$$\forall n \geq 1 : (A, B)^{(1,n)} \models \mathsf{E}h(B^1) \Rightarrow (A, B)^{(1,n+1)} \models \mathsf{E}h(B^1)$$

Proof. In order to prove the lemma, we only need to consider the case where the left-hand side of the implication is true, i.e., cases where the smaller system satisfies the specification. Consider a run x of the small system that satisfies the specification. We construct a corresponding run y in the larger system: We set $y(A^1)$ to $x(A^1)$ and $y(B^i)$ to $x(B^i)$ for all $i \in [1, n]$. Moreover, we we let the additional process B^{n+1} in y stutter in its initial state init_B. Because the additional stuttering run $y(B^{n+1})$ does not change the behaviour of the remaining runs, and we copied all local runs of the original system to the larger system, y still satisfies the specification. □

Lemma 5.1.2 (Disjunctive Bounding Lemma [35])
For all closed disjunctive process templates (A, B)*,*

$$\forall n \geq c : (A,B)^{(1,n)} \models \mathsf{E}h(A^1) \Leftrightarrow (A,B)^{(1,c)} \models \mathsf{E}h(A^1) \text{ where } c = |B| + 1 \tag{5.1.1}$$

$$\forall n \geq c : (A,B)^{(1,n)} \models \mathsf{E}h(B^1) \Leftrightarrow (A,B)^{(1,c)} \models \mathsf{E}h(B^1) \text{ where } c = |B| + 2 \tag{5.1.2}$$

If $c < n$, the Disjunctive Bounding Lemma allows us to reduce the number of B-processes from n to c without changing the system's behaviour regarding the desired specification. We first cover the proof for the second part of the lemma, and then describe the required modifications in order to prove the first part.

Proof. $\Longrightarrow$ In the following, we construct a run y in the cutoff system that is equivalent to a given run $x = (s_1, e_1, p_1)\ldots$ in the larger system. Here, $y(A^1)$ and $y(B^1)$ are set to the corresponding local runs $x(A^1)$ and $x(B^1)$ without any modification up to stuttering. If x is an infinite run, y must also be infinite. If $x(A^1)$ or $x(B^1)$ is infinite, then y is already infinite. Otherwise, there exists some index i such that $x(B^i)$ is an infinite local run. Then, we set $y(B^c) = x(B^i)$.

In order to enable all transitions along $y(A^1)$ and $y(B^1)$ (and possibly $y(B^c)$), we need to guarantee that all corresponding guards are enabled. To this end, consider a non-trivial (i.e., non-empty) guard g labelling a transition $s_j(B^1) \to s_{j+1}(B^1)$. Moreover, consider some state t that is necessary in order to enable g. Because g is enabled in the j-th step of x, there is at least one other process whose local state is t in the j-th step. If $t \in T_A$, the global state in the j-th step of y already contains t, therefore g is enabled in the larger system. Otherwise, there exists an index i such that $s_j(B^i) = t$. In the latter case, we need to introduce a local run in y which ensures that state t is present in the j-th step. Let MinComp(t) be the shortest prefix of some local computation in x leading to state t. Moreover, let the function MinLen(t) be the length of MinComp(t). Applied to the desired state $t \in T_B$, MinComp(t) is a prefix of some B-process' local path. Since MinComp(t) is the prefix of the run that first visits t, there does not exist any transition in x whose guard is enabled by a local copy of t for any step d within $0 \leq d <$ MinLen(t). Therefore, we can ensure that all guards of transitions in x requiring t are enabled by adding a *flooding run* for t to y, i.e., a run which first approaches t in the same way as MinComp(t), and then stutters in t for an infinite number of steps if x is infinite, and for $|x| -$ MinLen(t) steps if x is finite. More generally, let $Reach \subseteq T_B$ be the set of B states that are visited at least once in x. We add a flooding path for each such state. To this end, we assign the local run of $|Reach|$ process instances as follows. Let $f : [1, |Reach|] \to Reach$ be a bijection. Then, we set $y(B^{i+1}) =$ MinComp$(f(i)) \circ (f(i))^l$, where l is ω if y is infinite, $|y| -$ MinComp$(f(i))$ otherwise. We let all remaining B-processes in y stutter in their initial states (which is allowed by the monotonicity lemma).

Consider two states $t, t' \in T_B$ and their corresponding flooding runs $y(B^i)$ and $y(B^m)$. If both flooding runs are based on a single local run of x, i.e., MinComp(t) is a prefix of MinComp(t'), the interleaving constraint for asynchronous systems is violated. This violation can be corrected by adding additional steps and let the processes move sequentially. That is, we insert a stuttering step l after each step j, where both instances B^i and B^m take a transition. We then set $y_l(B^i) = y_{j+1}(B^i)$, and $y_l(B^m) = y_j(B^m)$. If n process instances are involved in the violation, we insert $n - 1$ additional stuttering steps, and split up the single global transition into n global transitions, each corresponding to a single local transition.

This run construction shows that for each path x in the first system, there exists a corresponding path y in the second system. Since $y(B^1)$ is equivalent to $x(B^1)$ except for additional stuttering steps, y fulfills the LTL\X formula $h(B^1)$ iff x does.

$\Longleftarrow$ We repeatedly apply the Disjunctive Monotonicity Lemma and add $n - c$ processes, which stutter in their initial states.

□

The proof for the first part of the lemma is similar. Here, we do not need to preserve the computation $x(B^1)$, thus the cutoff is reduced by 1 if the specification considers A^1 instead of B^1.

Lemma 5.1.3 (Disjunctive Truncation Lemma [35])
$\forall n_A, n_B : (A, B)^{(n_A,n_B)} \models \mathsf{E}h(B^1) \Leftrightarrow (A, B)^{(m_A,m_B)} \models \mathsf{E}h(B^1)$ *where* $m_A = \min(n_A, |A| + 1)$ *and* $m_B = \min(n_B, |B| + 2)$

Proof. If $n_B \leq |B| + 2$, $n_B = m_B$. Otherwise, we let A' be the parallel composition of the n_A instances of A, i.e., $A' = (A)^{(n_A)}$. Then, $(A, B)^{(n_A,n_B)} = (A', B)^{(1,n_B)}$ and $(A', B)^{(1,n_B)} \models \mathsf{E}h(B^1)$ iff $(A', B)^{(1,|B|+2)} \models \mathsf{E}h(B^1)$ (Disjunctive Bounding Lemma) iff $(A, B)^{(m_A,|B|+2)} \models \mathsf{E}h(B^1)$. Similarly, if $n_A \leq |A|+1$, $n_A = m_A$. Otherwise, let $A' = (B)^{(m_B)}$ and $B' = A$. Then, $(A, B)^{(m_A,m_B)} \models \mathsf{E}h(B^1)$ is equivalent to $(A', B')^{(1,n_A)} \models \mathsf{E}h(A^1)$, which further is satisfied iff $(A', B')^{(1,|A|+1)} \models \mathsf{E}h(A^1)$ (Disjunctive Bounding Lemma) iff $(A, B)^{(m_A,m_B)} \models \mathsf{E}h(B^1)$. [35] □

Theorem 5.1.4 (Disjunctive Cutoff Result [35])
Let f *be* $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$ *or* $\bigwedge_{i\in[1,n_k]} \mathsf{E}h(\mathcal{T}_k^i)$.

$$\forall(n_1, n_2) \succeq (1, 1) : (\mathcal{T}_1, \mathcal{T}_2)^{(n_1,n_2)} \models f \Leftrightarrow \forall(d_1, d_2) \preceq (c_1, c_2) : (\mathcal{T}_1, \mathcal{T}_2)^{(d_1,d_2)} \models f$$

where c_l *is the cutoff for the* l*-th template,* $c_k = |\mathcal{T}_k| + 2$ *and for* $l \neq k : c_l = |\mathcal{T}_l| + 1$.

We restrict the set of formulas f we need to consider in the proof. Without loss of generality, we assume that $f = \mathsf{E}h(\mathcal{T}_2^1)$ because of three reasons. First, consider specifications f with the structure $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$. Since all n_k instances of template k are isomorphic, $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$ is equivalent to $\mathsf{A}h(\mathcal{T}_k^i)$ for any $i \in [1, n_k]$. Second, we can use the duality of E and A to convert each A specification into an E specification by negating h. Third, $\mathsf{E}h(\mathcal{T}_1^1)$ is equivalent to $\mathsf{E}h(\mathcal{T}_2^1)$ if we reorder the templates, i.e., swap the template indices. These simplifications result in the following formula.

$$\forall(n_1, n_2) \succeq (1, 1) : (\mathcal{T}_1, \mathcal{T}_2)^{(n_1,n_2)} \models \mathsf{E}h(\mathcal{T}_2^1) \text{ iff } \forall(d_1, d_2) \preceq (c_1, c_2)(\mathcal{T}_1, \mathcal{T}_2)^{(d_1,d_2)} \models \mathsf{E}h(\mathcal{T}_2^1)$$
$$\text{where } c_l = |\mathcal{T}_l| + l$$

Proof. $\Longrightarrow$ Consider any $n_1, n_2 \geq 1$, and let $m_1 = \min(n_1, |\mathcal{T}_1| + 1)$, $m_2 = \min(n_2, |\mathcal{T}_2| + 2)$. By the Disjunctive Truncation Lemma, $(\mathcal{T}_1, \mathcal{T}_2)^{(n_1,n_2)} \models \mathsf{E}h(\mathcal{T}_2^1)$ iff $(\mathcal{T}_1, \mathcal{T}_2)^{(m_1,m_2)} \models \mathsf{E}h(\mathcal{T}_2^1)$. Thus, for all systems with size $(n_1, n_2) \succeq (|\mathcal{T}_1| + 1, |\mathcal{T}_2| + 2)$ it is enough to show that the specification is satisfied in systems of size $(|\mathcal{T}_1| + 1, |\mathcal{T}_2| + 2)$. $\Longleftarrow$ If the specification holds for all (n_1, n_2), it also holds for all $(d_1, d_2) \preceq (c_1, c_2)$. □

Disjunctive Monotonicity Lemma, Disjunctive Bounding Lemma, and Disjunctive Truncation Lemma can easily be generalized to an arbitrary number of templates. For the sake of notational clarity, we omit repeating the multi-template versions of these theorems. Instead, we only provide the final cutoff result for the PMCP of disjunctive cutoff systems given single-indexed properties.

Theorem 5.1.5 (Disjunctive Cutoff Theorem [35])
Let f *be* $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$ *or* $\bigwedge_{i\in[1,n_k]} \mathsf{E}h(\mathcal{T}_k^i)$. *Then,*

$$\forall(n_1, \ldots, n_k) \succeq (1, \ldots, 1) : (\mathcal{T}_1, \ldots, \mathcal{T}_k)^{(n_1,\ldots,n_k)} \models f \textit{ iff}$$
$$\forall(d_1, \ldots, d_k) \preceq (c_1, \ldots, c_k)(\mathcal{T}_1, \ldots, \mathcal{T}_k)^{(d_1,\ldots,d_k)} \models f$$

where c_l *is the cutoff for the* l*-th template,* $c_k = |\mathcal{T}_k| + 2$, *and* $\forall l \neq k : c_l = |\mathcal{T}_l| + 1$.

Theorem 5 in [35] deals with the cutoffs for model-checking against double-indexed properties (proofs are omitted).

Theorem 5.1.6 (Disjunctive Cutoff Theorem for Process Pairs [35])
Let f *be* $\bigwedge_{i\in[1,n_k],j\in[1,n_l]} \mathsf{A}h(\mathcal{T}_k^i, \mathcal{T}_l^j)$ *or* $\bigwedge_{i\in[1,n_k],j\in[1,n_l]} \mathsf{E}h(\mathcal{T}_k^i, \mathcal{T}_l^j)$. *Then,*

$\forall(n_1,\ldots,n_k) \succeq (1,\ldots,1) : (\mathcal{T}_1,\ldots,\mathcal{T}_k)^{(n_1,\ldots,n_k)} \models f$ *iff*
$\forall(d_1,\ldots,d_k) \preceq (c_1,\ldots,c_k)(\mathcal{T}_1,\ldots,\mathcal{T}_k)^{(d_1,\ldots,d_k)} \models f$

where $c_k = |\mathcal{T}_k| + 2$, $c_l = |\mathcal{T}_l| + 2$, *and for* $\forall m \neq k, l : c_m = |\mathcal{T}_m| + 1$.

We take a closer look at the required modifications to the existing lemmas in order to support double- and multi-indexed properties. The Disjunctive Monotonicity Lemma (Lemma 5.1.1) allows adding processes to a given system without violating its behaviour regarding the specification. We observe that this lemma still holds if we consider multi-indexed properties, since we let the additional processes stutter in the initial state and do not change any of the existing local runs. By contrast, the Disjunctive Bounding Lemma (Lemma 5.1.2) explicitly uses the structure of single-indexed properties. We therefore introduce an adapted version of the Disjunctive Bounding Lemma that supports multi-indexed properties.

Lemma 5.1.7 (Disjunctive Bounding Lemma (for Multi-Indexed Properties) [7])
For all closed disjunctive process templates (A, B),

$$\forall n \geq c : (A,B)^{(1,n)} \models \mathsf{E}h(A^1, B^{(k)}) \Leftrightarrow (A,B)^{(1,c)} \models \mathsf{E}h(A^1, B^{(k)}) \text{ where } c = |B| + k + 1$$

Proof Idea $\Longrightarrow$ As in the original proof, we construct a run y of the cutoff system for each possible run x of the large system. Properties that do not include any B-processes have no influence on the cutoff for template B. As a result, $k = 0$ corresponds to the first part of Lemma 5.1.2. Moreover, the case where $k = 1$ is equivalent to the second part of Lemma 5.1.2. Here we need to preserve $x(A^1)$ as well as $x(B^1)$ when constructing a cutoff run for a given run x of large system. For $k \geq 2$, we need to preserve the runs of k processes of template B. □

The final multi-template theorem for multi-indexed properties can be shown similarly as for single-indexed properties.

5.2 Conjunctive Cutoffs

In this section we describe the cutoff proofs for conjunctive guarded systems. As for disjunctive guards, we first introduce the required lemmas and then prove the Conjunctive Cutoff Result, i.e., the cutoff theorem for systems with two templates. Finally, we generalize the Conjunctive Cutoff Result to support systems with an arbitrary number of templates.

Lemma 5.2.1 (Conjunctive Monotonicity Lemma [35])
For all closed conjunctive process templates (A, B),

$\forall n \geq 1 : (A,B)^{(1,n)} \models \mathsf{E}h(A^1) \Rightarrow (A,B)^{(1,n+1)} \models \mathsf{E}h(A^1)$
$\forall n \geq 1 : (A,B)^{(1,n)} \models \mathsf{E}h(B^1) \Rightarrow (A,B)^{(1,n+1)} \models \mathsf{E}h(B^1)$

Proof. Let x be a run of the smaller system that satisfies the specification. We build a corresponding run y of the larger system which also satisfies the specification. To this end, we set $y(A^1) = x(A^1)$ and $\forall i \in [1,n] : y(B^i) = x(B^i)$. Moreover, we let the additional process B^{n+1} stutter in its initial state. This additional local computation does not change the enabledness of transitions along other local runs due to the restriction that conjunctive guards must contain the guard variables corresponding to each template's initial state. Thus, transitions along y are not blocked by the additional process. Because $y(A^1)$ and $y(B^1)$ are copies of $x(A^1)$ and $x(B^1)$, respectively, the larger system is equivalent to the smaller system regarding the specification. [35] □

Lemma 5.2.2 (Conjunctive Bounding Lemma [35, 7])
For all closed conjunctive process templates (A, B)*,*

$\forall n \geq c : (A, B)^{(1,n)} \models \mathsf{E}h(A^1) \Leftrightarrow (A, B)^{(1,c)} \models \mathsf{E}h(A^1)$ *where* $c = 2|B| - 2$
$\forall n \geq c : (A, B)^{(1,n)} \models \mathsf{E}h(B^1) \Leftrightarrow (A, B)^{(1,c)} \models \mathsf{E}h(B^1)$ *where* $c = 2|B| - 2$

For simplicity, we first prove the second part of the lemma, i.e., we consider the specification $\mathsf{E}h(B^1)$, and then describe the necessary modifications for the proof of the first part. [1]

Proof. $\Longrightarrow$ Let $x = (s_1, e_1, p_1) \ldots$ be a run of the larger system that satisfies the specification. We construct a corresponding run y of the cutoff system, where $y(A^1) = x(A^1)$ and $y(B^1) = x(B^1)$ up to stuttering. Thus, the specification is also satisfied by y. Since for conjunctive guarded systems, adding a process to the already existing system results in equally or more strengthened guards, we do not need additional local runs for y in order to enable the transitions along $y(A^1)$ and $y(B^1)$. The two local runs in y are copies of the original system, and thus the guards are enabled in the smaller system if they are also enabled in the original system. Additionally, for infinite runs x where both $x(A^1)$ and $x(B^1)$ are finite (i.e, locally deadlocked), we need to construct an infinite corresponding run y. In this case, there exists an infinite local computation $x(B^i)$. Thus, we set $y(B^2) = x(B^i)$.

To exclude the case that the large system deadlocks, but the cutoff system satisfies $\mathsf{E}h(B^1)$, we must consider the case where x is globally deadlocked. There exist at most $|B| - 1$ states (i.e., all states except the initial state) which disable the guards of all successor transitions of local states $s_j(A^1)$ and $s_j(B^1)$ in some deadlock state s_j. We extend y by selecting one local run for each T_B state that is necessary to reproduce the deadlock of x. However, in order to ensure a global deadlock, we must ensure that these copied computations are also deadlocked. Otherwise, the deadlock can be easily resolved by letting non-deadlocked processes move out of the particular "bad" state. Consider a transition $t' \xrightarrow{g} t$ of some process, where $t' \neq \text{init}_B$ and g is such that it is enabled by all states of T_A and T_B except for t'. Because guards are not reflexive, i.e., they do not consider the process whose guard is evaluated, the particular stuttering process is allowed to take the particular transition. This state transition is possible since there is no other process which is in state t' during the guard evaluation. In order to prevent that any transition of the additional run is enabled when the original run deadlocks, we need at most one additional local computation for each state of $T_B \setminus \{\text{init}_B\}$, which finally stutters in the particular state.

$\Longleftarrow$ We repeatedly apply the monotonicity lemma and let the remaining processes stutter in their initial states. □

The cutoff under consideration of infinite runs is at most $c = 2$. If x is infinite and $x(A^1)$ or $x(B^1)$ is infinite, $c = 1$, otherwise $c = 2$ because we need to copy one infinite local run $x(B^i)$ to y. In order to reproduce deadlocked runs, we need at most 2 local runs for each T_B state except for the initial state, i.e. $2(|B| - 1)$ local runs. Consider the case where B^1 is deadlocked in state init_B. At first sight, still $\leq 2(|B| - 1)$ additional runs are required in order to ensure a global deadlock. Some transition $t \xrightarrow{g} t'$ is the last transition in y, i.e., the one that leads to the global deadlock in step j. Note the following facts.

- Because the global state already contains at least one of all bad T_B states, process A^1 is deadlocked. Thus, this transition belongs to some B-process.
- The transition is not unguarded, otherwise there is no global deadlock in step j.
- The global state contains at least two copies of t, one copy of t', and at least one copy of all other bad T_B states.

[1] Note that Emerson and Kahlon [35] proved the cutoff to be $c = 2|B|$ for the first case, and $c = 2|B| + 1$ for the second case. However, we consider refined cutoffs from [7].

As a result, there does not exist a non-trivial, and enabled guard g in this step, and the cutoff is $2(|B|-1)$.

Consider the first part of the Conjunctive Bounding Lemma. Because the specification only includes A^1, we do not need to preserve the local computation $x(B^1)$ as described for the second part. Thus, the newly constructed corresponding path y contains only one local computation of x, namely $y(A^1) = x(A^1)$. For infinite computations x of the original system where $x(A^1)$ is locally deadlocked, we set $y(B^1)$ to an infinite computation $x(B^i)$ up to stuttering. Thus, the cutoff for the number of B instances is $c = 1$ in this case. However, if x is deadlocked, we need to ensure that all outgoing transitions from the last state of A^1 are disabled in the same deadlock step j as in the original run (up to stuttering). To this end, we add flooding paths for at most $|B| - 1$ states (i.e., all states of B except the initial state). Because of the guards' irreflexivity we require at most $2(|B| - 1)$ such runs. Thus, the overall cutoff for deadlocked systems is $c = 2(|B| - 1) = 2|B| - 2$.

Lemma 5.2.3 (Conjunctive Truncation Lemma [35])
For all closed conjunctive templates (A, B)*,*

$$\forall n_A, n_B : (A, B)^{(n_A, n_B)} \models \mathsf{E}h(B^1) \Leftrightarrow (A, B)^{(m_A, m_B)} \models \mathsf{E}h(B^1)$$

where $m_A = \min(n_A, 2|A| - 2)$ *and* $m_B = \min(n_B, 2|B| - 2)$

The proof is similar to the proof of the Disjunctive Truncation Lemma.

Theorem 5.2.4 (Conjunctive Cutoff Result [35])
Let f *be* $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$ *or* $\bigwedge_{i\in[1,n_k]} \mathsf{E}h(\mathcal{T}_k^i)$*.*

$$\forall (n_1, n_2) \succeq (1, 1) : (\mathcal{T}_1, \mathcal{T}_2)^{(n_1,n_2)} \models f \Leftrightarrow \forall (d_1, d_2) \preceq (c_1, c_2) : (\mathcal{T}_1, \mathcal{T}_2)^{(d_1,d_2)} \models f$$

where $c_l = 2|\mathcal{T}_l| - 2$ *is the cutoff for the* l*-th template.*

The proof is similar to the proof of the Disjunctive Cutoff Result.

Theorem 5.2.5 (Conjunctive Cutoff Theorem [35])
Let f *be* $\bigwedge_{i\in[1,n_k]} \mathsf{A}h(\mathcal{T}_k^i)$ *or* $\bigwedge_{i\in[1,n_k]} \mathsf{E}h(\mathcal{T}_k^i)$*. Then,*

$$\forall (n_1, \ldots, n_k) \succeq (1, \ldots, 1) : (\mathcal{T}_1, \ldots, \mathcal{T}_k)^{(n_1,\ldots,n_k)} \models f \Leftrightarrow$$
$$\forall (d_1, \ldots, d_k) \preceq (c_1, \ldots, c_k)(\mathcal{T}_1, \ldots, \mathcal{T}_k)^{(d_1,\ldots,d_k)} \models f$$

where c_l *is the cutoff for the* l*-th template, and* $c_l = 2|T_k| - 2$*.*

The proof of this lemma is similar to the proof of the Disjunctive Cutoff Theorem.

Chapter 6

Parameterized Synthesis for Guarded Systems

In order to reduce the parameterized template synthesis problem to a standard template synthesis problem, we aim at using the cutoffs of Emerson and Kahlon [35] described in Chapter 5. To this end, we need to tackle two problems. First, Emerson and Kahlon only consider closed systems. However, the goal of synthesis is usually to construct a system whose behaviour can be influenced by the environment, i.e., an open system. Second, using the cutoffs introduced in Chapter 5 does not allow synthesizing liveness properties because they do not support any notion of fairness. This chapter describes our theoretical work consisting of two solutions. The first solution focuses on obtaining an open system by synthesizing a closed system and converting it into an open one (Section 6.1). This solution allows to directly use the existing cutoffs, however, does not address fairness, and is thus only applicable for synthesizing safety properties. Moreover, it suffers from state explosion. For this reason, we follow a second approach that is based on a thorough analysis and extension of the cutoffs from [35], in order to support both open systems and liveness properties (Section 6.2).

6.1 Conversion from Open to Closed Systems

As already pointed out, we cannot directly use the existing cutoffs for the synthesis of open systems. A solution which allows us to directly use the existing cutoff results is to synthesize closed systems. Given a specification Φ, we interpret all signals provided by the environment as outputs instead of inputs of the particular processes, and let the processes non-deterministically choose the assignments for these variables. We then synthesize closed templates $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ such that $(\mathcal{T}_1, \ldots, \mathcal{T}_k)^{(n_1,\ldots,n_k)} \models \Phi$, and convert each of the resulting closed system templates into an equivalent open template. Figure 6.1 illustrates the general idea. This approach entails the following requirements. First, there must be a representative closed system for each possible open system. This representative system must behave equivalently to the desired open system regarding its states and outputs. Second, there must be a unique open system for each closed system such that the closed system resulting from synthesis can be translated to the final open system without any ambiguities.

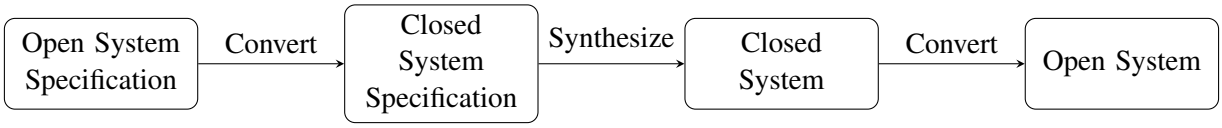


Figure 6.1: Obtaining open systems by synthesizing closed systems

6.1.1 Closed System Types

Consider an open process template $\mathcal{T} = (T, \text{init}, \delta, \text{guard}, o)$ with inputs E and outputs O. A corresponding closed system $\mathcal{T}'$ with inputs $E' = \emptyset$ and $O' = O \times E$ models the environment's choice of values for each variable in E by a non-deterministic selection of the successor state. We define the corresponding closed system to be an LTS $\mathcal{T}' = (T', \text{init}', \delta', o')$. The state is a tuple containing both the original open LTS state and the selected input values, i.e, $T' \subseteq T \times \mathbb{B}^E$. Likewise, the output of each state is a tuple that consist of the original output and the inputs associated to the particular state. Consider some closed system state $(t, e) \in T'$. We call the component representing the original open LTS state the *state component*, and denote the second part as *input component*.

The increased number of output variables in the closed system yields a larger set of observable states because the set of guard variables in the closed system is $G' = \bigcup_{k \in [1,k]} \mathbb{B}^{O_k \cup E_k}$. Therefore, we need to prevent divergence of the closed system's behaviour from the original open system's behaviour. This means that in closed system guards all guard variables corresponding to a single open system guard variable must have the same valuation. To this end, we introduce a guard modification function mod_guard : $\mathbb{B}^G \to \mathbb{B}^{G'}$ that maps an open system transition guard $g \in \mathbb{B}^G$ into a corresponding closed system transition guard $g \in \mathbb{B}^{G'}$ as follows. For each guard variable that is true (false) in g, all corresponding closed system guard variables (that is, closed system guard variables associated to the same valuation of variables in O_k as the open system guard variable) are set to true (false) in the resulting closed system guard. Note that there is a bijective mapping between closed system guards obtained by applying mod_guard and the set of open system guards, which is a prerequisite for converting closed systems into open systems.

There exist two different closed system models. The difference between them is the semantics of the states' input component.

Last Input Closed Systems In a *Last Input Closed System* (LICS), we interpret the input component of a state as the previously processed input. For example, consider an open template transition $t \xrightarrow{e'} t'$. Then, the corresponding closed LTS transition is $(t, e) \to (t', e')$, where e is the environment input which is consumed by the transition that connects the predecessor of t and t.

We convert an open template $\mathcal{T} = (T, \text{init}, \delta, \text{guard}, o)$ into a LICS template $\mathcal{T}' = (T', \text{init}', \delta', \text{guard}', o')$ according to the following conversion rules.

1. For each original state $t \in T$ and each incoming transition labelled with input valuation e, there exists a state $(t, e) \in T'$ in the corresponding closed template.
2. For each state $(t, e) \in T'$, the output component of $o'((t, e))$ must be equal to $o(t)$.
3. For each transition $t_j \xrightarrow{e,g} t_{j+1}$ in the open template, and for each $t'_j \in T'$ corresponding to t_j (i.e., with state component t), there exists a transition $t'_j \xrightarrow{g'} (t_{j+1}, e)$ with $g' = \text{mod\_guard}(g)$.

These rules ensure that each environment input that causes a transition to be enabled can be "generated" by the closed system. The output behaviour of the closed system matches the output behaviour of the original system according to the second rule. As described by the third rule, we add for each transition labelled with guard g in the original template a set of transitions labelled with guard g' in the new template. Here, guard g' is retrieved by applying the guard modification function to g.

Figure 6.2a shows a small open process template with state set $T = \{t_0, t_1, t_2\}$, initial state t_0, and two input bits. The corresponding LICS obtained by conversion is shown in Figure 6.2c. There is a single state that represents the original initial state t_0, however, since t_1 of the open LTS is reachable via two transitions with different inputs (00, 01), the conversion also yields a separate t_1 state for each input. In the initial closed system state, the current "input" is determined by choosing either the transition that

leads to $(t_1, 00)$ or the transition that leads to $(t_1, 01)$. For both states corresponding to t_1, there is only a single successor which is reachable by choosing one specific input value 01 (as in the original template). Consider Figure 6.2b and Figure 6.2c. The example shows the similarity between input-preserving open systems and LICS. Both store the information about the last input in their states. The only difference is that input-preserving open systems receive their inputs from the environment, while LICS determine inputs by choosing transitions non-deterministically. Thus, their transitions are not labelled with input values.

Next Input Closed Systems In the second closed process template type, we interpret the input component of a state as the current (or next, respectively) input. Given the open template transition $t \xrightarrow{e} t'$, the corresponding closed template transition is $(t, e) \to (t', e')$, where e' is the input consumed by the transition between t' and its successor. These closed systems are called *Next Input Closed Systems* (NICS).

The conversion of an open process template $\mathcal{T}$ into a NICS template $\mathcal{T}'$ happens according to the following rules.

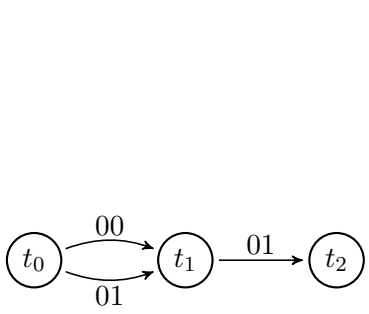
1. For each open system state $t \in T$ and each outgoing transition labelled with input assignment $e \in \mathbb{B}^E$, there exists a closed system state $(t, e) \in T'$.
2. For each state $(t, e) \in T'$, the output component of $o'((t, e))$ must be equal to $o(t)$.
3. For each open system transition $t_j \xrightarrow{e,g} t_{j+1}$, and for all $t'_{j+1} \in T'$ with state component t_{j+1}, there exists a closed system transition $(t_j, e) \xrightarrow{g'} t'_{j+1}$ with $g' = \text{mod\_guard}(g)$.

Figure 6.2d shows a closed process template that is obtained by converting the open template in Figure 6.2a into a NICS.

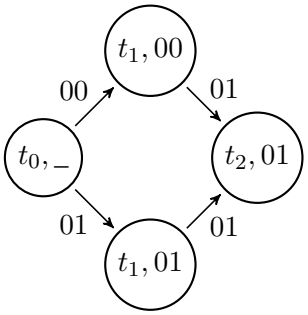
The construction rules for both closed template types ensure that for each path in the open template $p = (t_1, e_1)(t_2, e_2)\ldots$, there exists a corresponding state sequence in the closed template ($p_{\text{LICS}} = (t_1, \epsilon)(t_2, e_1)\ldots$, and $p_{\text{NICS}} = (t_1, e_1)(t_2, e_2)\ldots$, respectively). Moreover, the closed template's trace is equivalent to the open template's trace for each path. This allows us to replace a parallel composition of multiple open systems by an equivalent composition of closed systems. Consider a system type $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$. We replace the first process template by a corresponding NICS or LICS $\mathcal{T}'_1$. Because of the trace equivalence between open and closed systems, each global state of the original system and all corresponding global states of the new system (i.e., with one template replaced) are mapped to the same state guard. Furthermore, each closed template transition is labelled with a guard that is equivalent to the particular open transition's guard. As a result, we have the following properties for each computation of a system based on the original system type $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, and its corresponding computation of the system based on the modified system type $(\mathcal{T}'_1, \mathcal{T}_2, \ldots, \mathcal{T}_k)$. Local paths of $\mathcal{T}_1$ processes are replaced by equivalent closed local transitions (of $\mathcal{T}'_1$). Here, the enabledness of each transition along the new local path is equivalent to the enabledness of the particular transition in the original path. Local paths of processes that are based on other than the replaced template remain unchanged. This allows an iterative replacement of each open template $\mathcal{T}_k$, until all templates in the system type are closed ones. By applying each conversion rule in the opposite direction, a closed process template is converted into an equivalent open template. Based on iterative replacement of each closed process template by the equivalent open template, we retrieve an open guarded system.

6.1.2 Issues

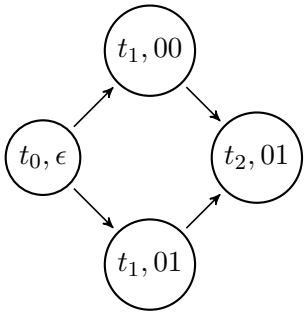
The approach of synthesizing closed systems and converting them into open systems allows us to directly use the cutoffs for synthesizing guarded systems. However, this solution has several drawbacks. The



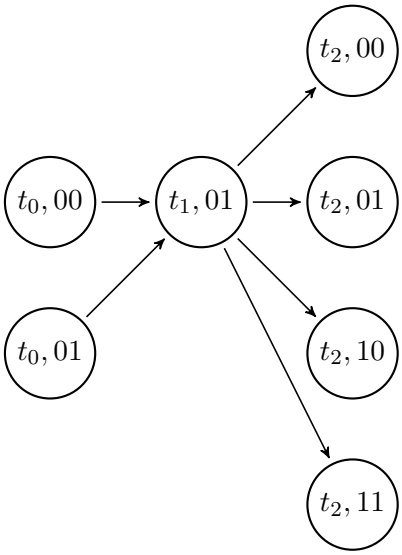
(a) Open LTS



(b) Input-Preserving Open LTS



(c) Last Input Closed LTS



(d) Current Input Closed LTS

Figure 6.2: Examples for different LTS types

conversion from open to closed systems yields a state explosion, that is, converting an open template with states T_k and inputs E_k results in a closed template with up to $|T_k| \times |\mathbb{B}^{E_k}|$ states. As a consequence, we need to synthesize a closed template with $\leq |T_k| \times |\mathbb{B}^{E_k}|$ states. The impact to synthesis is twofold. On the one hand, we need to consider templates that are bigger than the desired open system, by a factor that is exponential in the number of input variables. On the other hand, the cutoffs described in Chapter 5 directly depend on the number of template states. Consequently, not only the number of states, but also the cutoffs (number of required instances) increase exponentially in the size of the desired open template. Furthermore, the original cutoffs do not consider the aspect of fairness, which is, as mentioned in Chapter 3, an absolute necessity for the synthesis of liveness properties. Hence, by our conversion solution we can only obtain systems that satisfy given safety properties.

6.2 Cutoffs for Synthesis

Modelling open systems as closed systems (see Section 6.1) has the major disadvantage that it results in a state explosion. Furthermore, we can only synthesize safety properties with this approach because it does not consider any type of fairness. To tackle these problems, we adapt the existing cutoffs in order to support synthesizing fair open systems directly. The remainder of this section is organized as follows. First, we recapitulate the proof structure of the existing cutoff proofs. Then, we focus on making these cutoffs applicable to open systems. In the next step, we analyze possible fairness constraints and select an appropriate one. Based on the aspect of fairness, we finally provide new cutoffs for the synthesis of fair open systems.

6.2.1 Proof Structure

The analysis of the proofs in [35] shows that the proof structure is similar for disjunctive and conjunctive guard cutoffs. Besides stuttering bisimulation of a large system and the corresponding cutoff system, the following concepts are essential [7].

Symmetry The cutoffs are for systems that consist of multiple isomorphic process groups. For this reason, it is enough to consider only one process for single-indexed properties, and two processes for double-indexed properties, respectively.

- $\bigwedge_i \mathsf{E}h(\mathcal{T}_k^i)$ iff $\mathsf{E}h(\mathcal{T}_k^1)$
- $\bigwedge_{i,j:i\neq j} \mathsf{E}h(\mathcal{T}_k^i, \mathcal{T}_k^j)$ iff $\mathsf{E}h(\mathcal{T}_k^1, \mathcal{T}_k^2)$
- $\bigwedge_{i\in[1,n_k]} \bigwedge_{j\in[1,n_l]} \mathsf{E}h(\mathcal{T}_k^i, \mathcal{T}_l^j)$ iff $\mathsf{E}h(\mathcal{T}_k^1, \mathcal{T}_l^1)$

This equivalence allows us to prove the monotonicity lemmas and bounding lemmas for both guard types under consideration of the simplified specification.

Monotonicity Lemma If a specification $\mathsf{E}h(\mathcal{T}_1^1)$ or $\mathsf{E}h(\mathcal{T}_2^1)$ is satisfied by a system $(\mathcal{T}_1, \mathcal{T}_2)^{(1,n)}$, it is also satisfied by the system $(\mathcal{T}_1, \mathcal{T}_2)^{(1,n+1)}$. This is shown by constructing a corresponding path in the larger system for each path in the smaller system. To this end, the additional process $\mathcal{T}_2^{n+1}$ stutters in the initial state infinitely often. For disjunctive guarded systems, adding a path does not disable any guard, because extending the global state by an additional local state "weakens" guards (i.e., in the new system, there are possibly more guards enabled). By contrast, enlarging the global state of conjunctive guarded systems "strengthens" guards (i.e., there possibly exist guards that are enabled in the original system, but not in the new system). However, processes that are in the respective initial states do not disable any conjunctive guard by definition.

Bounding Lemma If a specification $\mathsf{E}h(\mathcal{T}_1^1)$ or $\mathsf{E}h(\mathcal{T}_2^1)$ is satisfied by a system $(\mathcal{T}_1, \mathcal{T}_2)^{(1,n)}$, then the specification is satisfied by the cutoff system $(\mathcal{T}_1, \mathcal{T}_2)^{(1,c)}$. The proof is based on showing that for each run in the original (larger) system there exists a stuttering-equivalent run in the cutoff system. For disjunctive guarded systems, the constructed run consists of a) all local runs which are contained in the specification and b) one additional local run for each state that is reachable in the original run, in order to ensure that all transitions which are enabled in the original run are also enabled in the constructed run. These additional local runs consist of the shortest path to the particular state, and an (infinite) stuttering sequence. The bounding lemma for conjunctive guarded systems (in [35]) distinguishes between infinite and globally deadlocked runs. The proof for the infinite case is based on removing local runs that are not contained in the specification. The global deadlock cutoff is shown by constructing a global deadlock in the cutoff system based on the global deadlock in the large system.

Cutoff Result The Bounding Lemma is extended to both templates by template reordering and using a parallel composition of one template's instances (Truncation Lemma). The final Cutoff Result for two templates is proved by using the duality between E and A as well as the symmetry constraint. The Cutoff Theorem is a generalization of the Cutoff Result for an arbitrary number of processes. This derivation is equivalent for both guard types.

6.2.2 Cutoffs for Open Systems

In Section 6.1 we described that for each open system there exists a corresponding closed system, that is, for each run in an open system, there exists an equivalent run in the corresponding closed system such that the sequence of inputs provided by the environment to the open system is modeled by the closed system using non-determinism. Therefore, the existence of a path in the closed system entails both a sequence of inputs and a path for the corresponding open system. If this input sequence is provided to the particular open system, all transitions along the desired computation are enabled.

Hence, each run that satisfies a specification in the closed system is also realizable in the open system, and vice versa. Because the proofs of Emerson and Kahlon are based on a stuttering bisimulation, and do not include assumptions regarding characteristics of closed systems, they can also be used to reason about open system runs instead of closed system runs. Thus, we can use the existing cutoff results for open systems directly.

6.2.3 Fairness Considerations

As mentioned in Chapter 3, certain properties of a specification cannot be satisfied without any assumptions regarding the environment. For example, consider the liveness property $\bigwedge_i \mathsf{G}(\mathsf{F}(s(\mathcal{T}^i) = t_0) \wedge \mathsf{F}(s(\mathcal{T}^i) = t_1))$, which should be satisfied by a system $(\mathcal{T})^{(n)}$ with states $T = \{t_0, t_1, \dots\}$. In other words, the requirement is that both states t_0 and t_1 are visited infinitely often by all process instances. Obviously, the property can easily be violated by the environment, e.g. if at least one process is never scheduled. This "bad" scheduling must be avoided. Hence, the scheduling is essential for the existence of a specified system: If one of the processes is never scheduled, the specification is violated, and no valid system can be synthesized. We introduce a fair scheduling constraint which restricts the set of possible schedulings to the relevant (i.e., fair) ones. This constraint is added to each property of the specification as follows.

$$(\text{assumption} \wedge \text{fair\_scheduling}) \rightarrow \text{guarantee}$$

The left-hand side of the implication is a conjunction of assumptions given in the property and the fairness constraint. The right-hand side is the guarantee of the property that needs to be satisfied by the system. Adding a fairness constraint to our example property yields the following formula:

$$\text{fair\_scheduling} \rightarrow \left[\bigwedge_i \mathsf{G}(\mathsf{F}(s(\mathcal{T}^i) = t_0) \wedge \mathsf{F}(s(\mathcal{T}^i) = t_1))\right]$$

Note that the set of "bad" schedulings strongly depends on the system architecture. Fairness constraints must be defined such that "bad" schedulings are avoided, however, without forbidding "good" (fair) schedulings. Furthermore, fairness constraints complicate the specification, leading to an increased complexity of the synthesis problem. Thus, the constraint must be as compact as possible. We analyze the following fairness constraints. [1]

- Unconditional fairness [9]: $\bigwedge_p \mathsf{GF}\,\text{moves}_p$

[1] Note that in our system model, we do not allow the scheduling of processes that are not enabled. Under this assumption, weak fairness [9] (every process is scheduled infinitely often) coincides with unconditional fairness, and we do not consider it separately.

- Strong fairness [9]: $\bigwedge_p (\mathsf{GF}\,\text{enabled}_p \rightarrow \mathsf{GF}\,\text{moves}_p)$

Here, enabled_p is true iff there is at least one enabled transition for process p. For open systems, the enabledness depends on both the guards and the environment input. The signal moves_p is true iff process p is scheduled (which implies enabledness, by definition of our system model).

We use $\mathsf{E}_{\text{uncond}}h$ ($\mathsf{A}_{\text{uncond}}h$) to describe that a property h holds for at least one (for all) unconditionally fair runs.

$$\mathsf{E}_{\text{uncond}}h := \left[\bigwedge_p \mathsf{GF}\,\text{moves}_p\right] \rightarrow \mathsf{E}h$$

$$\mathsf{A}_{\text{uncond}}h := \left[\bigwedge_p \mathsf{GF}\,\text{moves}_p\right] \rightarrow \mathsf{A}h$$

$\mathsf{E}_{\text{strong}}h$ ($\mathsf{A}_{\text{strong}}h$) are defined likewise. Usually, we want to synthesize specifications under unconditional fairness. However, this constraint suffers from the fact that the process' enabledness does not only depend on the environment (i.e., the inputs) and the global state, but also on the implementation. This allows the synthesis algorithm to easily satisfy any property under fairness by falsifying the fairness constraint. For example, one such resulting implementation is a trivially deadlocked system with processes for which enabled_p is always false.

We need to counteract this problem by ensuring that there are no local deadlocks, i.e., $\bigwedge_p \mathsf{GF}\,\text{enabled}_p$. Obviously, deadlock freedom cannot be guaranteed unconditionally. It rather requires that each process is scheduled infinitely often. To this end, we weaken the desired property by adding the weakest fairness assumption we discussed above, i.e., strong fairness.

$$\bigwedge_p (\mathsf{GF}\,\text{enabled}_p \rightarrow \mathsf{GF}\,\text{moves}_p) \rightarrow \bigwedge_p \mathsf{GF}\,\text{enabled}_p$$

As a result, applying the enabledness constraint and the actual fairness constraint to a specification Φ results in the following formula.

$$\left(\bigwedge_p (\mathsf{GF}\,\text{enabled}_p \rightarrow \mathsf{GF}\,\text{moves}_p) \rightarrow \bigwedge_p \mathsf{GF}\,\text{enabled}_p\right) \wedge \left(\bigwedge_p \mathsf{GF}\,\text{moves}_p \rightarrow \Phi\right)$$

6.2.4 Cutoffs for Fair Guarded Systems

Based on our observations regarding the proof structure in Section 6.2.1, we first briefly point out the issues when considering open and fair systems. In the following two subsections we then derive new cutoffs that support open systems and fairness. For each guard type, we first introduce the Monotonicity Lemma, followed by the Bounding Lemma for infinite runs, and the Deadlock Detection Lemma. Because Truncation Lemma, Cutoff Result, and Cutoff Theorem under fairness are equivalent to the corresponding originals, we forego repeating them. For the sake of notational simplicity, we abbreviate the system type $(\mathcal{T}_1, \mathcal{T}_2)$ with (A, B).

Monotonicity Lemma Obviously, the proof structure of the monotonicity lemma for both guard types does not preserve fairness, since the construction used in the proof is based on letting additional processes stutter in their initial states (see Section 6.2.1).

Bounding Lemma In the disjunctive case, the cutoff run construction requires flooding runs, each one stuttering for a possibly infinite number of steps in one particular state that is reachable in the large run. This flooding construction obviously violates the fairness constraint. By contrast, the conjunctive bounding lemma part considering infinite systems preserves (unconditional) fairness. However, unconditional fairness is not applicable to the global deadlock part.

Challenges The cutoff proofs of Emerson and Kahlon do not support any kind of fairness. Deadlocks are not considered in case of disjunctive guards, and only partially (global deadlocks only) in case of conjunctive guards. For ensuring deadlock freedom, we need to detect both global and local deadlocks. Therefore, we introduce separate lemmas which prove that (local) deadlock freedom under strong fairness in the cutoff system implies (local) deadlock freedom under strong fairness in the large system. Because we use different fairness constraints for deadlock freedom and specification-specific properties, we introduce two counterparts for each original bounding lemma, namely a) a bounding lemma for specification properties under consideration of infinite systems (unconditional fairness), and b) a deadlock detection lemma (with respect to strong fairness). This structure also enables the modular use of different cutoffs in synthesis [52].

Cutoffs for Fair Disjunctive Guarded Systems

Lemma 6.2.1 (Disjunctive Monotonicity Lemma for Fair Open Systems [7])
For all open disjunctive process templates (A, B)*,*

$$\forall n \geq 1 : (A,B)^{(1,n)} \models \mathsf{E}_{uncond} h(A^1) \Rightarrow (A,B)^{(1,n+1)} \models \mathsf{E}_{uncond} h(A^1)$$
$$\forall n \geq 1 : (A,B)^{(1,n)} \models \mathsf{E}_{uncond} h(B^1) \Rightarrow (A,B)^{(1,n+1)} \models \mathsf{E}_{uncond} h(B^1)$$

Proof. Given an unconditionally fair run x of the original system that satisfies the specification $h(A^1)$ ($h(B^1)$, respectively), we construct a corresponding fair run y of the larger system. We set $y(A^1) = x(A^1)$ and $\forall i \in [1, n] : y(B^i) = x(B^i)$. For the additional process, $y(B^{n+1}) = x(B^n)$. Note that the local path duplication leads to the violation of the interleaving constraint because $y(B^n)$ and $y(B^{n+1})$ move simultaneously. We resolve the interleaving by adding an additional step whenever both processes move. In each such step, we first let process B^n move, while B^{n+1} and all other processes remain in their current states. In the next step, process B^{n+1} moves and all other processes including B^n remain in their respective previous states. The inputs of stuttering processes must not change by our system model definition. Then, each transition along $y(B^{n+1})$ is enabled if the corresponding transition along $y(B^n)$ is enabled. Because all local runs of y are copies of the local runs in x up to stuttering, all transitions that are enabled along local runs of x are also enabled along local runs of y. □

Lemma 6.2.2 (Disjunctive Bounding Lemma for Fair Open Systems [7])
For all open disjunctive process templates,

$$\forall n \geq c : (A,B)^{(1,n)} \models \mathsf{E}_{uncond} h(A^1) \text{ iff } (A,B)^{(1,c)} \models \mathsf{E}_{uncond} h(A^1), \text{ where } c = 2|B|$$
$$\forall n \geq c : (A,B)^{(1,n)} \models \mathsf{E}_{uncond} h(B^1) \text{ iff } (A,B)^{(1,c)} \models \mathsf{E}_{uncond} h(B^1), \text{ where } c = 2|B|$$

For the sake of clarity, we first prove the first part of the lemma and then consider the second part.

Proof. $\Longrightarrow$ Let $x = (s_0, e_0, p_0)\ (s_1, e_1, p_1) \ldots$ be an unconditionally fair run of the original system $(A, B)^{(1,n)}$ satisfying the specification $h(A^1)$. We construct a corresponding fair run y of the cutoff system $(A, B)^{(1,c)}$. In order to preserve the behaviour of the local run contained in the specification, $x(A^1)$ is copied to y modulo stuttering. As in the original proof (see Lemma 5.1.2), we introduce a flooding path for each state in T_B that is eventually reached in the run x (see Section 5.1). Each flooding path consists of the shortest local path in x from init_B to the particular flooded state, followed by infinitely many stuttering steps. Because infinite stuttering paths violate the fairness constraint, we modify the original proof. To this end, consider the set of flooded states $Reach$, which is the union of the following two disjoint subsets: a) $Reach_{\text{fin}}$, the set of states in T_B that are visited finitely often, and b) $Reach_{\text{inf}}$, the set of states in T_B that are visited infinitely often in the original run. For each set, we provide a specific flooding path construction. Consider some $t \in Reach_{\text{fin}}$. Let B^f be the process that first reaches t in step j, i.e., $s_j(B^f) = t$, and let B^g be the process that is the last one to leave t in some step m, i.e., $m = \max(j \mid \exists i : s_j(B^i) = t)$. Then, the flooding path of $t \in Reach_{\text{fin}}$ is the concatenation

$x(B^f)$ until step j and the infinite suffix $x(B^g)$ beginning at step m. By definition, this construction is fair because both parts of the new path are contained in the original fair run.

We now consider the set of infinitely often visited states $Reach_{\text{inf}}$. Let t be some state in this set, and B^f be the process that first reaches t in step j. Then, the flooding path for this state consists of $x(B^f)$ until step j, followed by an infinite stuttering sequence in state t. In order to make the path fair, we interrupt the infinite stuttering sequence and let the particular process move out and return again to t. Since T_B is finite and t is visited infinitely often, there is such a *loop path* in the original run x. However, we need to ensure that all transitions along this loop path from t to t are enabled. By the flooding constructions, all states $Reach_{\text{inf}} \setminus \{t\}$ are present in the global state. However, if the loop path contains a transition that requires some process to be in state t, we need another process in t while the particular flooding process moves along the loop path. We distinguish between the following cases.

1. The flooding path of t consists of a single self-transition guarded with t.
2. There is some other flooded state t' whose loop path contains t, i.e., t and t' are part of the same SCC.

In the first case, we need two flooding paths for each flooded state t. In the latter case, we obviously do not need an additional flooding path for each of the two states t and t'. Instead, we introduce a single copy of the flooding path for t' (w.l.o.g.), and use it as a *shared flooding path*. We then interrupt the infinite stuttering as follows. Consider some step in which the three processes corresponding to the flooding paths are in their particular flooded states, i.e., t, t', and t', respectively. First, we let the additional process move along the loop path from t' to t (third computation). Then, we let the process in state t move to t' (first computation). Finally, we let the remaining process (second computation) move from t' to t. The resulting local states of the three processes are t', t, t. Note that in each step where one process moves along the loop path, the other two processes are in t and t', respectively. Naturally, processes corresponding to other flooding paths are in the particular flooded state. Thus, all transitions along the loop paths are enabled.

$\Longleftarrow$ We apply the Disjunctive Monotonicity Lemma for fair systems $n - c$ times. □

Compared to the original Disjunctive Bounding Lemma, preserving fairness yields at most one additional process for each SCC of infinitely often visited states. Let n_{SCC} denote the minimal number of SCCs in B such that all reachable states are in at least one SCC. Then, the cutoff for the first part of the lemma is $c = |B| + n_{\text{SCC}}$. In the worst case, $Reach_{\text{inf}} = T_B$, and all loop paths consist of a self-transition. As a result, the cutoff for specifications of the form $\mathsf{E}_{\text{uncond}}(A^1)$ is $c = 2|B|$.

Consider the second part of the lemma. Here, we need to preserve the local run $x(B^1)$. Hence, we set $y(B^1) = x(B^1)$ (and also $y(A^1) = x(A^1)$ as for the first part). As before, we need at most $|B|$ flooding paths in order to enable all transitions along $x(A^1)$ and $x(B^1)$, and additional flooding path copies in order to preserve fairness. Because $x(B^1)$ is infinite, there is some step, after which the process corresponding to $x(B^1)$ visits only states in $Reach_{\text{inf}}$, i.e., the process moves along an SCC of $Reach_{\text{inf}}$ states infinitely often. This SCC contains at least one state (i.e., self-loop), and at most $Reach_{\text{inf}}$ states. In the worst case regarding the cutoff, the SCC consists of a single state $t \in Reach_{\text{inf}}$. Then, $x(B^1)$ is a flooding path for t. As a result, $c = 1 + (2|Reach_{\text{inf}}| - 1) \leq 2|B|$. If the SCC contains more than one state, $x(B^1)$ is a shared flooding path for all SCC states. Let $t, t' \in Reach_{\text{inf}}$ be visited by $x(B^1)$ infinitely often. Whenever process $x(B^1)$ is in state t, we let the process corresponding to the flooding path for t move along the loop path back to t. Next, we let process $x(B^1)$ continue moving until it visits state t'. Then, the process corresponding to the flooding path for t' moves along the self loop and back to t', and so forth. Consequently, the cutoff for this case is strictly less than $2|B|$.

Thus, the cutoff for the second part of the lemma is equal to the cutoff for the first part of the lemma, i.e., $c = 2|B|$.

Lemma 6.2.2 only considers infinite runs. Because we want synthesize deadlock-free systems, we need to ensure that if deadlock-freedom holds for a cutoff system, any larger system is also deadlock-free. To this end, we introduce the Disjunctive Deadlock Detection Lemma.

Lemma 6.2.3 (Disjunctive Deadlock Detection Lemma for Fair Open Systems [7])
For strong-fair runs of open systems $(A, B)^{(1,n)}$, $c = 2|B| - 1$ *is a cutoff for local deadlock detection.*

Proof. $\Longrightarrow$ For each strong-fair and locally deadlocked run $x = (s_1, e_1, p_1)\ (s_2, e_2, p_2)\ldots$ of the system $(A, B)^{(1,n)}$, there is a locally deadlocked strong-fair run in the cutoff system $(A, B)^{(1,c)}$. Let j denote the step in run x in which the system has at least one locally deadlocked process.

Let $\text{Visited}_\perp = \text{Visited}^1_\perp \cup \text{Visited}^2_\perp$ be the set of locally deadlocked states along x. $\text{Visited}^1_\perp$ contains all states that are only deadlocked if there is no other process in the same state, and $\text{Visited}^2_\perp$ contains all states that are deadlocked although there is some other process in the same state. $P^1_\perp$ ($P^2_\perp$) contains the set of processes p for which $s_j(p) \in \text{Visited}^1_\perp$ ($s_j(p) \in \text{Visited}^2_\perp$). We define $\text{Visited}_{\text{inf}}$ to be the set of infinitely often visited states, and $\text{Visited}_{\text{fin}}$ to be the set of states that are visited finitely often by processes not in $P^1_\perp$. We set $y(A^1) = x(A^1)$, $y(p) = x(p)$ for all $p \in P^1_\perp$, and define the remaining local runs of B-processes as follows. We add one flooding process for each state in $\text{Visited}_{\text{fin}} \cup \text{Visited}^2_\perp$, and two flooding processes for each state in $\text{Visited}_{\text{inf}}$.

By definition, processes in $\text{Visited}_{\text{fin}}$ must either deadlock or eventually move into some infinitely often visited state. We let the particular processes in y imitate the behaviour of the corresponding processes in x. Fairness for non-deadlocked processes is ensured by introducing a self-loop path sequence for the particular processes.

The resulting construction yields the cutoff $\text{Visited}_{\text{fin}} + \text{Visited}_\perp + \text{Visited}_{\text{inf}} \leq 2|B|$. Further investigations show that the cutoff is strictly less than $2|B|$. There are three different cases of local deadlocks.

1. Process A^1 deadlocks because of the absence of some state $t \in T_B$. Obviously, there neither exists a process which deadlocks in t nor a process which visits t infinitely often. Thus, t is in $\text{Visited}_{\text{fin}}$, visited by some process $p \in \text{Visited}^1_\perp$, or not visited at all. Consequently, the cutoff construction results requires at most two processes for $T_B \setminus \{t\}$, and at most one process for state t.

2. A B-process is deadlocked in some state $t \in \text{Visited}^2_\perp$. Thus, y contains one flooding process for t and at most 2 flooding processes for each state except for t.

3. A B-process is deadlocked in some state $t \in \text{Visited}^1_\perp$. If $t \in \text{Visited}_{\text{fin}}$, the process which does not deadlock in t leaves t and eventually moves into some state $t'' \in \text{Visited}^2_\perp \cup \text{Visited}_{\text{inf}}$. If $t'' \in \text{Visited}^2_\perp$, $c \leq 2|B| - 1$. Otherwise, we omit one of the two flooding processes for t'', (does not violate strong fairness) and thus $c \leq 2|B| - 1$.

$\Longleftarrow$ Because $n > |B|$, there is at least one process p which either moves infinitely often, or is deadlocked in some state in $\text{Visited}^2_\perp$. We repeatedly apply the monotonicity lemma, and let the additional processes imitate the behaviour of process p. □

Cutoffs for Fair Conjunctive Guarded Systems

The proof of the original Conjunctive Monotonicity Lemma is based on letting the additional process remaining in the initial state forever, which does not influence (i.e., disable) any guards. Such a construction obviously violates unconditional fairness. Thus, the additional process must move infinitely often. However, in case of conjunctive guarded systems, we cannot, for example, let the new process move along the same path as one of the other processes (as in case of the Lemma 6.2.1), since the additional path possibly causes a (local) deadlock. We tackle the problem by restricting the set of considered conjunctive guarded systems to the set of *initializing conjunctive guarded systems*, where each B-process visits the initial state infinitely often. The restriction is represented as an additional guarantee in the specification that must be satisfied under unconditional fairness.

$$\bigwedge_{i \in [1, n_{\mathrm{B}}]} \mathsf{A}_{\text{uncond}}\, \mathsf{GF}\, \mathrm{s}(B^i) = \mathrm{init}_B$$

Runs that satisfy the additional property are *unconditionally initializing*. Adding such a restriction is reasonable, because it holds for a plethora of protocols, like the AMBA bus protocol and various cache coherence protocols.

Lemma 6.2.4 (Conjunctive Monotonicity Lemma for Fair Open Systems [7])
For all unconditional-fair initializing runs of open conjunctive guarded systems,

$$\forall n \geq 1 : (A, B)^{(1,n)} \models \mathsf{E}_{\mathit{uncond}} h(A^1) \Rightarrow (A, B)^{(1,n+1)} \models \mathsf{E}_{\mathit{uncond}} h(A^1)$$
$$\forall n \geq 2 : (A, B)^{(1,n)} \models \mathsf{E}_{\mathit{uncond}} h(B^1) \Rightarrow (A, B)^{(1,n+1)} \models \mathsf{E}_{\mathit{uncond}} h(B^1)$$

Proof. Let x be a unconditional-fair run of the smaller system. We construct a run y in the larger system based on x. To this end, we set $y(A^1) = x(A^1)$ and $\forall i \in [1, n] : y(B^i) = x(B^i)$. By definition of initializing runs, $x(B^n)$ visits the initial state infinitely often. We take advantage of this property and split $x(B^n)$ into a sequence of infinitely many path slices, where each sequence starts in the initial state init_B. In y, we let process B^n move along $x(B^n)$ until $x_j(B^n) = \mathrm{init}_B$ at some time j. This sequence corresponds to the first path slice. In other words, $\forall m < j : y_m(B^n) = x_m(B^n), y_m(B^{n+1}) = \mathrm{init}_B$. Then, we let process B^n stay in the initial state, while process B^{n+1} moves along the second path slice. After B^{n+1} enters the initial state again, we let process B^n move along the third path slice, and so forth. The existence of infinitely many path slices implies that there are infinitely many such alternations. Thus, the resulting runs $y(B^n)$ and $y(B^{n+1})$ satisfy unconditional fairness (processes move infinitely often) and are initializing (the initial state is visited infinitely often). Because all other local runs are copies of the original local runs up to stuttering, y is initializable and satisfies unconditional fairness. □

Lemma 6.2.5 (Conjunctive Bounding Lemma for Fair Open Systems)
For all unconditional-fair initializing runs of open conjunctive guarded systems,

$$\forall n \geq 1 : (A, B)^{(1,n)} \models \mathsf{E}_{\mathit{uncond}} h(A^1) \text{ iff } (A, B)^{(1,1)} \models \mathsf{E}_{\mathit{uncond}} h(A^1)$$
$$\forall n \geq 2 : (A, B)^{(1,n)} \models \mathsf{E}_{\mathit{uncond}} h(B^1) \text{ iff } (A, B)^{(1,2)} \models \mathsf{E}_{\mathit{uncond}} h(B^1)$$

Proof. $\Longrightarrow$ Let x be a run of the original system that satisfies all properties assumed by the lemma. We construct a corresponding run y in the cutoff system. In order to preserve the system's behaviour regarding the specification, we set $y(A^1) = x(A^1)$ and $y(B^1) = x(B^1)$ in the first case, and additionally $y(B^2) = x(B^2)$ in the second case. All transitions along the local runs are enabled if they are enabled in the particular original local runs, because removing local runs weakens conjunctive guards. y is infinite because x is infinite, and we do not need additional adaptations.

$\Longleftarrow$ We repeatedly apply the Conjunctive Monotonicity Lemma for Fair Open Systems. □

Note that the $\Longrightarrow$ direction of the proof yields a cutoff of $(0,1)$, $(1,0)$ respectively. However, in order to apply the monotonicity lemma, we need for each template at least one process not occurring in the specification. Therefore, the cutoff is $(1,1)$, or $(1,2)$, depending on the specification.

By the new Conjunctive Bounding Lemma, there exists an unconditional-fair run in the cutoff system for each unconditional-fair run in the original system that satisfies a certain specification, and vice versa. We also need to ensure that deadlock-freedom in the cutoff system implies deadlock-freedom in any larger system. Therefore, we introduce cutoffs for deadlock detection. Note that the proof of Lemma 5.2.2 considers global deadlocks. We can directly apply this partial result to fair systems, because strong fairness is trivially satisfied in any case if there is a global deadlock.

Lemma 6.2.6 (Conjunctive Global Deadlock Detection Lemma for Fair Open Systems [35, 7])
For strong-fair systems, $2|B|-2$ is a cutoff for global deadlock detection on strong-fair runs.

Since we not only want to detect and avoid global deadlocks, but also local deadlocks, we extend our results by the following lemma.

Lemma 6.2.7 (Conjunctive (Local) Deadlock Detection Lemma for Fair Open Systems [7])
For strong-fair, 1-guard conjunctive systems, $2|B|-2$ is a cutoff for (local) deadlock detection on strong-fair runs.

Proof. $\Longrightarrow$ Let $x = (s_1, e_1, p_1)\ (s_2, e_2, p_2)\ \ldots$ be a deadlocked run of the large system, where each non-deadlocked (infinite) local run is strong-fair and initializing. That is, after some step j, x contains at least one deadlocked local run. Let $P_\perp$ be the set of processes that are deadlocked in x, and $\text{Visited}_\perp = \text{Visited}^1_\perp \cup \text{Visited}^2_\perp$ be the set of final states of processes in $P_\perp$.

1. The set $\text{Visited}^2_\perp$ contains deadlock states t, which have at least one transition guarded with the complement of $\{t\}$. That is, t requires one other process in state t in order to be disabled.
2. The set $\text{Visited}^1_\perp$ contains deadlock states, which do not require another process to be in the same state in order to be disabled.

Moreover, let $\text{Visited}_{\text{inf}}$ be the set of infinitely often visited states. By definition, $\text{Visited}^1_\perp$, $\text{Visited}^2_\perp$, and $\text{Visited}_{\text{inf}}$ are disjoint. By BlockSet we denote the set of states which are required to disable all transitions of states in $\text{Visited}_\perp$.

We set $y(A^1) = x(A^1)$ and for each $t \in \text{BlockSet} \cap T_B$:

1. If $t \in \text{Visited}^1_\perp$, there exists at least one local run $x(B^i)$ such that $s_j(B^i) = t$. We copy this particular local run to y.
2. If $t \in \text{Visited}^2_\perp$, there exist at least two local runs with local state t in step j. We copy these two local runs to y.
3. If $t \in \text{Visited}_{\text{inf}}$, we can distinguish between two cases.
   (a) x contains either an infinite run, which eventually only visits t, i.e., it moves along a self-loop from t to t. In this case, we let one B-process in y imitate the behaviour of the particular process in x.
   (b) There are at least two processes B^i, B^m, such that $s_j(B^i) = s_j(B^m) = t$. Moreover, B^i and B^m are part of a set of processes that visit t infinitely often in x. In each step, at least one of these processes is in state t. Each process that visits t infinitely often, moves along (some) loop path from t to t. Obviously, this loop path does not contain transitions labelled with $\neg\{t\}$. In order to ensure that t is present in y after step j, we construct two local runs as follows. We copy the prefixes of $x(B^i)$ and $x(B^m)$, and then let the two processes alternately move along a loop path from t to t. While one process moves, the other one needs to stutter in t.

$\Longleftarrow$ We repeatedly apply the monotonicity lemma. □

$|\text{BlockSet} \cap T_B| < |T_B|$ since init_B does not disable any transition by definition. Since we need at least two processes for each state in BlockSet, the cutoff is $2|B| - 2$.

Without the restriction to 1-guard systems, the BlockSet consists of sets of states (subsets of $\mathcal{P}(T_B)$. In each step, there is possibly a different blocking set responsible for ensuring the particular (local) deadlocks, which does not allow a static construction as above. [2]

Table 6.1 summarizes the bounding lemma cutoff results presented here, in [35], and in [7]. The left-hand side of the table shows the cutoff results without considering fairness. Here, disjunctive and conjunctive property cutoffs proved by Emerson and Kahlon were applied without any modifications. In [7], we consider deadlock detection without fairness for both guard types, and extend the results of [35] who only consider global deadlocks for conjunctive guards. For local deadlock detection in disjunctive guarded systems, we currently do not know a smaller cutoff than $2|B| - 1$ (which can be proved in the same way as for the fair case). Note that the local deadlock cutoff for non-fair conjunctive guards (marked with (*)) is only valid for the subset of initializing systems. The right-hand side of the table shows the cutoff results with respect to fairness, which are subject of this work and [7]. The new cutoff results for disjunctive guards are valid unconditionally. By contrast, property cutoffs for conjunctive guards only apply to initializing systems. Fair conjunctive deadlock detection (local deadlock detection, to be precise [7]) is applicable for 1-guard systems only (marked with (**)).

	Without fairness		**With fairness**	
	Property $h(A^1, B^k)$ [35, 7]	Deadlock Detection [7]	Property $h(A^1, B^k)$ [7]	Deadlock Detection [7]
Disjunctive	$\|B\| + k + 1$	global: $\|B\| + 2$ local: $2\|B\| - 1$	$\max(2\|B\|, 2\|B\| + k - 1)$	$2\|B\| - 1$
Conjunctive	$k + 1$	global: $2\|B\| - 2$ [35] local: $2\|B\| - 2$ (*)	$k + 1$ (*)	$2\|B\| - 2$ (**)

Table 6.1: Cutoff results for modular synthesis of fair systems [7]

[2] Due to the results, we restrict ourselves to 1-guard systems, since finding a cutoff for general conjunctive systems is outside the scope of this work.

Chapter 7

Algorithm

For synthesizing guarded systems, we use the distributed bounded synthesis algorithm [69], which was already described in Section 3.6.2. In order to add support for guarded systems, we need to adapt certain parts of the original algorithm, under consideration of the following aspects.

- Fairness Constraints: Before translating the given specification into a UCT automaton, we need to add fairness constraints for each liveness property as well as additional properties used as system assumptions in the cutoff proofs.
- Process Templates: We synthesize a predefined number of different process templates instead of a set of isomorphic processes.
- Interleaving Constraints: In each global step of the desired system, only one process is allowed to move.
- Guards: We need to introduce a guard assignment function that assigns a guard to each transition as well as a guard evaluation function that determines the enabledness of each transition in the system with respect to the global state.

In Section 7.1, we first describe the modified procedure in Algorithm 2 informally, and will go into more detail in the following subsections. In Section 7.2 we then discuss possible optimizations, including applicable techniques from other publications that use the bounded synthesis approach.

7.1 Bounded Synthesis of Guarded Systems

Algorithm 2 shows the main steps of the modified bounded synthesis approach. As an input, the algorithm gets the parameterized specification Φ, the desired numbers of instances n_k and the initial template size b_k for each process template with index k as well as a value max_increments that restricts the number of bound increments. This maximum number of loop iterations is required to ensure the semi-algorithm's termination. The algorithm returns the implementations of all process templates, or UNSAT if there are no templates $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})$ with $(b_1, \ldots, b_\mathrm{k}) \preceq (|T_1|, \ldots, |T_\mathrm{k}|) \preceq (b_1 + \text{max\_increments}, \ldots, b_\mathrm{k} + \text{max\_increments})$ such that the system $(\mathcal{T}_1, \ldots, \mathcal{T}_\mathrm{k})^{(n_1,\ldots,n_\mathrm{k})}$ satisfies the specification. As described in Chapter 3, liveness properties must hold whenever fairness is provided, while safety properties must hold unconditionally (with respect to fairness). We assume that the specification does not include any fairness constraints. Hence, the algorithm first processes the specification (Line 4) and extends each liveness property by the unconditional fairness assumption introduced in Chapter 6. Moreover, architecture specific constraints Φ_{arch} are added to the specification. In the succeeding loop (Line 10), we first determine the minimum number of instances required for synthesizing a

correct system with respect to the given multiplicity $(n_1, \ldots, n_k)$. To this end, we calculate the cutoff for each property of the specification as well as the deadlock detection cutoff. Then, all properties of the specification are instantiated based on the property cutoffs. As in the original algorithm, we finally encode the bounded synthesis problem in SMT and use an SMT solver in order to determine whether there exists a solution for the given problem. If the SMT formula is satisfiable, the process template implementations are extracted from the model provided by the SMT solver. Otherwise, the loop is repeated until a solution is found for some increased bound values, or the number of loop iterations exceeds max_increments.

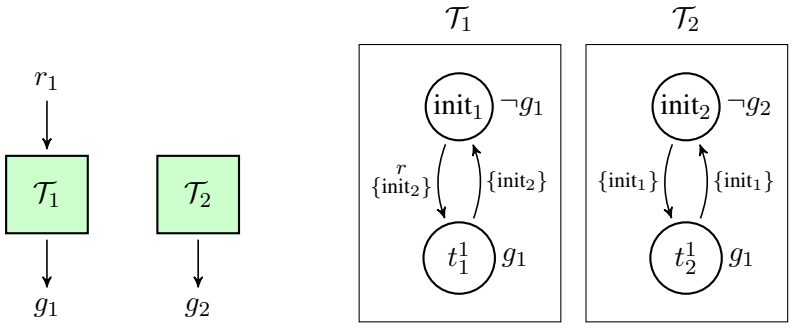
Data: Φ, $N = (n_1, \ldots, n_k)$, $b = (b_1, \ldots, b_k)$, max_increments
Result: $(\mathcal{T}_1, \ldots, \mathcal{T}_k)$ or UNSAT
$\Phi' \leftarrow \mathsf{true}$
foreach $\Phi_{prop} \in \Phi$ **do**
  if Φ_{prop} *is a liveness property* **then**
    $\Phi_{\text{prop}} \leftarrow (\text{fair\_scheduling} \rightarrow \Phi_{\text{prop}})$
  end
  $\Phi' \leftarrow \Phi' \wedge \Phi_{\text{prop}}$
end
$\Phi' \leftarrow \Phi' \wedge \Phi_{\text{arch}}$
increments $\leftarrow 0$
while *increments* $<$ *max _increments* **do**
  $C_p \leftarrow$ get_cutoffs(Φ', N, b)
  $\varphi \leftarrow$ instantiate(Φ', b, C_p)
  formula $\leftarrow$ encode(φ, b, C_p)
  res $\leftarrow$ solve(formula)
  if *res* $==$ *SAT* **then**
    return extract_model()
  end
  $b \leftarrow (b_1 + 1, \ldots, b_k + 1)$
  increments $\leftarrow$ increments $+ 1$
end
return UNSAT

Algorithm 2: Bounded Synthesis of Guarded Systems

In the following subsections, we discuss modifying the specification, detecting cutoffs, instantiating the specification and the final SMT encoding more precisely. To this end, we use the following simple mutual exclusion scenario as a running example. The desired system consists of two conjunctive guarded process templates (see Figure 7.1a). The first process template $\mathcal{T}_1$ has an input r_1, which allows the environment to request a shared resource, as well as an output g_1, which is raised whenever access to the resource is granted. The second process template $\mathcal{T}_2$ provides a grant signal g_2 with the same semantics as g_1, however, $\mathcal{T}_2$ has no input signals. Instead, the process template decides internally when to allow access to the resource. The parameterized specification is

$$\Phi := \left(\bigwedge_{i \in [1,n_1]} \mathsf{AG}(r_1^i \rightarrow \mathsf{F}(g_1^i)) \right) \wedge \left(\bigwedge_{i \in [1,n_2]} \mathsf{A}\,\mathsf{GF}(g_2^i) \right) \wedge \left(\bigwedge_{i \in [1,n_1]} \bigwedge_{j \in [1,n_2]} \mathsf{AG}\left(\neg(g_1^i \wedge g_2^j) \right) \right)$$

All request signals must finally be granted by $\mathcal{T}_1$ processes, and each $\mathcal{T}_2$ process must grant infinitely often. No two pairs of $\mathcal{T}_1$ and $\mathcal{T}_2$ processes must grant access to the resource at the same time. Pairs of processes based on the same template are allowed to grant at the same time. A valid implementation is shown in Figure 7.1b.



(a) Process Templates for Running Example

(b) Valid Implementation for Running Example

Figure 7.1: Running Example

7.1.1 Specification

The specification consists of properties as defined in Chapter 4, that is, special prenex-indexed CTL*\X formulas over input signals and output signals. Architecture-specific formulae like the fairness condition and the property for deadlock freedom also contain architecture-specific internal signals, which are introduced in order to express architecture-related states. In particular, the signal is_enabled_k^i signifies whether process $\mathcal{T}_k^i$ is enabled in its current local state (with respect to the global state and environment inputs). The signal moves_k^i is true iff process $\mathcal{T}_k^i$ is scheduled (in this case $\text{is\_enabled}_k^i = \mathsf{true}$ by definition of our system model). The SMT encoder replaces every occurrence of an internal signals by a corresponding predicate.

Regarding the assume $\rightarrow$ guarantee style, properties only consist of a guarantee part. Each such property is expected to be either a safety or a liveness property.[1] In order to determine its type, we first obtain the smallest possible instantiation for the particular indexed property and then convert it into a corresponding UCT. For Büchi automata, the property is a safety property if the corresponding Büchi automaton contains a rejecting node, which can only reach itself or other rejecting nodes, but cannot reach any non-rejecting nodes. Otherwise, the property is a liveness property [1]. A UCT corresponds to a safety property if all rejecting nodes are absorbing (that is, they contain no outgoing edges or a true-labelled self-loop). Otherwise, the UCT corresponds to a liveness property [51]. Each liveness property (liveness guarantee in our case) Φ_{prop} is extended by the unconditional fairness constraint.

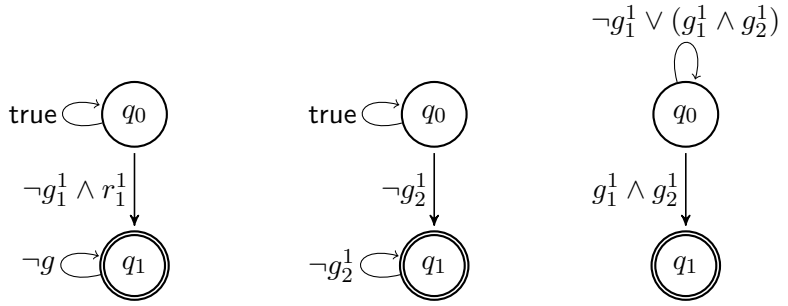
$$\Phi_{\text{prop}} := \left(\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \Phi_{\text{prop}}$$

Moreover, we define a specification part Φ_{arch} containing architecture-specific properties. For both disjunctive and conjunctive guarded systems, Φ_{arch} includes the architecture specific fairness property (see Section 6.2.3).

$$\Phi_{\text{arch}} := \left[\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \left(\mathsf{GF}\, \text{is\_enabled}_k^i \rightarrow \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{is\_enabled}_k^i \right]$$

Moreover, for synthesizing conjunctive guarded systems, we also need to ensure that the initial state is

[1] We assume that the specification defines a set of properties that are either safety or liveness properties. Note that each property containing both types can be split into two properties — one safety and one liveness property.



(a) UCT for guarantee $\mathsf{AG}(r_1^1 \rightarrow \mathsf{F}\, g_1^1)$ **(b)** UCT for guarantee $\mathsf{AGF}(g_2^1)$ **(c)** UCT for guarantee $\mathsf{AG}(\neg(g_1^1 \wedge g_2^1))$

Figure 7.2: UCTs for running example properties

visited infinitely often. Thus, we add the following property.[2]

$$\Phi_{\text{arch}} := \Phi_{\text{arch}} \wedge \left[\left(\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \left(\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{is\_init}_k^i \right) \right]$$

The resulting specification Φ can be seen as a conjunction of safety properties, (modified) liveness properties, and architecture properties.

$$\Phi := \Phi_{\text{safety}} \wedge \Phi_{\text{liveness}} \wedge \Phi_{\text{arch}}$$

Consider our running example. The UCTs for the smallest possible guarantee instantiations (that is, single-indexed properties instantiated for one, double-indexed properties instantiated for two processes) are shown in Figure 7.2. We observe that $\bigwedge_{i \in [1,n_1]} \mathsf{AG}(r^i \rightarrow \mathsf{F}\, g_1^i)$ and $\bigwedge_{i \in [1,n_2]} \mathsf{A}\,\mathsf{GF}\, g_2^i$ are liveness properties, whereas $\bigwedge_{i \in [1,n_1]} \bigwedge_{j \in [1,n_2]} \mathsf{AG}\left(\neg(g_1^i \wedge g_2^j)\right)$ is a safety property. Thus, the first two properties are extended by the fairness constraint, while the latter is left unchanged. Moreover, the previously added architecture specific properties are added. We obtain the following modified specification.

$$\begin{aligned}
\Phi := & \left[\bigwedge_{i \in [1,n_1]} \bigwedge_{j \in [1,n_2]} \mathsf{AG}\left(\neg(g_1^i \wedge g_2^j)\right) \right] \wedge \\
& \left[\left(\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \left(\left(\bigwedge_{i \in [1,n_1]} \mathsf{A}\,\mathsf{G}(r^i \rightarrow \mathsf{F}\, g_1^i) \right) \wedge \left(\bigwedge_{i \in [1,n_2]} \mathsf{A}\,\mathsf{GF}\, g_2^i \right) \right) \right] \wedge \\
& \left[\left(\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \left(\left(\bigwedge_{i \in [1,n_1]} \mathsf{A}\,\mathsf{GF}\, \text{is_init}_1^i \right) \wedge \left(\bigwedge_{i \in [1,n_2]} \mathsf{A}\,\mathsf{GF}\, \text{is_init}_2^i \right) \right) \right] \wedge \\
& \left[\bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \left(\mathsf{GF}\, \text{is_enabled}_k^i \rightarrow \mathsf{GF}\, \text{moves}_k^i \right) \rightarrow \bigwedge_{k \in [1,\text{k}]} \bigwedge_{i \in [1,n_k]} \mathsf{GF}\, \text{is_enabled}_k^i \right]
\end{aligned}$$

7.1.2 Cutoff Detection

The cutoff detection routine determines the deadlock detection cutoff as well as the cutoff required for each specified property. The calculation of the deadlock detection cutoff is trivial, since its value

[2] Note that this architectural property is also a liveness property. Therefore, it is also considered when applying operations to liveness properties in the following.

only depends on the current bounds. For calculating property cutoffs, the structure of each safety and liveness property's guarantee is considered. The cutoff of architecture properties is set to the previously determined deadlock detection cutoff.

Next, each resulting cutoff is compared element-wise to the desired number of instances $N = (n_1, \ldots, n_k)$. If the cutoff for a template is larger than the desired number of instances, we restrict the cutoff to the number of instances. This operation is intuitive: We do not want to use cutoffs which exceed the number of desired instances and thus increase the problem's complexity.

Under the assumption of bound $b = (2, 2)$, the cutoff detection yields an overall cutoff $c = (2, 2)$ cutoffs for our example, and constant cutoffs $(1, 2)$, $(2, 1)$, $(2, 2)$ for safety and liveness properties.

7.1.3 Specification Instantiation

Instantiating an indexed formula means replacing each quantified sub-formula h by a set of concrete instantiations, which are logically connected depending on the quantifier, i.e., as disjunctions in case of existential quantifiers, and as conjunctions in case of universal quantifiers. The instantiation results for supported properties are shown in the second column of Table 7.1.

Indexed Formula	Instantiation	
	without symmetry	with symmetry
$\bigwedge_{i\in[1,n_k]} \mathsf{A} h(\mathcal{T}_k^i)$	$h(\mathcal{T}_k^1) \wedge \cdots \wedge h(\mathcal{T}_k^{n_k})$	$h(\mathcal{T}_k^1)$
$\bigwedge_{i,j\in[1,n_k],i\neq j} \mathsf{A} h(\mathcal{T}_k^i, \mathcal{T}_k^j)$	$h(\mathcal{T}_k^1, \mathcal{T}_k^2) \wedge h(\mathcal{T}_k^2, \mathcal{T}_k^1) \wedge \cdots \wedge h(\mathcal{T}_k^{n_k-1}, \mathcal{T}_k^{n_k})$	$h(\mathcal{T}_k^1, \mathcal{T}_k^2)$
$\bigwedge_{i\in[1,n_k]} \bigwedge_{j\in[1,n_l]} \mathsf{A} h(\mathcal{T}_k^i, \mathcal{T}_l^j)$	$h(\mathcal{T}_k^1, \mathcal{T}_l^1) \wedge h(\mathcal{T}_k^1, \mathcal{T}_l^2) \wedge \cdots \wedge h(\mathcal{T}_k^{n_k}, \mathcal{T}_l^{n_l})$	$h(\mathcal{T}_k^1, \mathcal{T}_l^1)$

Table 7.1: Instantiation of indexed properties with and without using symmetry

By symmetry of isomorphic processes, we conclude that some single-indexed property $\bigwedge_i \mathsf{A} h(\mathcal{T}_k^i)$ holds if $\mathsf{A}\, h(\mathcal{T}_k^i)$ holds for some i, e.g. $\mathsf{A}\, h(\mathcal{T}_k^1)$. Similar considerations apply to double-indexed and other multi-indexed properties. This symmetry, which was already used in the proofs of the Disjunctive Cutoff Result (Theorem 5.1.4) and Conjunctive Cutoff Result (Theorem 5.2.4), decreases the size of property instantiations significantly, as shown in the third column of Table 7.1. Instead of conjuncts whose number is polynomial regarding the cutoff, instantiated properties consist of a single conjunct when using symmetry. We apply this optimization to all guarantees of safety and liveness properties. Note that symmetry considerations are not usable for quantified assumptions (e.g., the fairness constraint), because those must hold explicitly for all processes in the system. For this reason, we fully instantiate the left-hand side of liveness properties as well as universally quantified architecture properties for the given cutoff.

Instantiating the illustrative specification results in a CTL*\X formula, which is equivalent to the following LTL\X formula.

$$
\begin{aligned}
\varphi := \mathsf{G}\left(\neg(g_1^1 \wedge g_2^1)\right) \wedge\\
&\left[\left(\mathsf{GF}\,\text{moves}_1^1 \wedge \mathsf{GF}\,\text{moves}_1^2 \wedge \mathsf{GF}\,\text{moves}_2^1\right) \rightarrow \left(\mathsf{G}\left(r_1^1 \rightarrow \mathsf{F}\, g_1^1\right)\right)\right] \wedge\\
&\left[\left(\mathsf{GF}\,\text{moves}_1^1 \wedge \mathsf{GF}\,\text{moves}_2^1 \wedge \mathsf{GF}\,\text{moves}_2^2\right) \rightarrow \left(\mathsf{GF}\, g_2^1\right)\right] \wedge\\
&\left[\left(\mathsf{GF}\,\text{moves}_1^1 \wedge \mathsf{GF}\,\text{moves}_1^2 \wedge \mathsf{GF}\,\text{moves}_2^1\right) \rightarrow \left(\mathsf{GF}\,\text{is\_init}_1^1\right)\right] \wedge\\
&\left[\left(\mathsf{GF}\,\text{moves}_1^1 \wedge \mathsf{GF}\,\text{moves}_2^1 \wedge \mathsf{GF}\,\text{moves}_2^2\right) \rightarrow \left(\mathsf{GF}\,\text{is\_init}_2^1\right)\right] \wedge\\
&\Big[\big((\mathsf{GF}\,\text{is\_enabled}_1^1 \rightarrow \mathsf{GF}\,\text{moves}_1^1) \wedge (\mathsf{GF}\,\text{is\_enabled}_1^2 \rightarrow \mathsf{GF}\,\text{moves}_1^2) \wedge\\
&\quad (\mathsf{GF}\,\text{is\_enabled}_2^1 \rightarrow \mathsf{GF}\,\text{moves}_2^1) \wedge (\mathsf{GF}\,\text{is\_enabled}_2^2 \rightarrow \mathsf{GF}\,\text{moves}_2^2)\big) \rightarrow\\
&\quad \left(\mathsf{GF}\,\text{is\_enabled}_1^1 \wedge \mathsf{GF}\,\text{is\_enabled}_1^2 \wedge \mathsf{GF}\,\text{is\_enabled}_2^1 \wedge \mathsf{GF}\,\text{is\_enabled}_2^2\right)\Big]
\end{aligned}
$$

7.1.4 SMT Encoding

The SMT encoding function called in Algorithm 2 gets an instantiated formula φ, bounds b, and cutoffs C_p for each property of φ, and creates a formula that encodes the bounded synthesis problem in SMT. The resulting formula consists of an architecture specific part on the one hand, and of a specification part on the other hand. We first focus on the architecture specific part, and then describe how the specification specific part is encoded. Note that in our description, we denote the bit-wise conjunction by the symbol &, and the bit-wise disjunction by the symbol |.

Architecture Encoding

All data types and functions required for encoding guarded systems are defined in the architecture-specific part. Consider the set of bounds $b = (b_1, \ldots, b_k)$. For each template we define the following basic constructs.

- State data-type $T_k = (\text{init}_k, t_k^1, \ldots, t_k^{b_k-1})$,
- Uninterpreted guard assignment function $\text{guard}_k : T_k \times E_k \times T_k \to \mathbb{B}^G$
- Uninterpreted output assignment function $o : T_k \to \mathbb{B}$ for each output $o \in O_k$
- State guard assignment function $\text{guard\_bit}_k : T_k \to \mathbb{B}^G$ that determines which guard variable is set for a particular local state

As defined by the system model, the set of guard variables for label guards is $G = \bigcup_{k\in[1,k]} \mathbb{B}^{O_k}$. In our encoding, each guard variable is represented as a single bit of a $|G|$-sized bit vector. Figure 7.3 illustrates the bit vector partitioning. Each template $\mathcal{T}_k$ is assigned a chunk of $|\mathbb{B}^{O_k}|$ bits in increasing order (the template with the smallest index is assigned the chunk which includes the LSB). The guard_bit_k function is fully interpreted, and maps each observable state to a unique bit inside the bit chunk of the particular template. Let $O_k = \{o_1, \ldots, o_m\}$. Then,

$$\text{guard\_bit}_k(t) := 1 \ll \left[\left(\sum_{l=0}^{k-1} |O_l| \right) + \sum_{i=1}^{|O_k|} \left(\begin{cases} 1 \ll (i-1) & \text{if } o_i(t) \\ 0 & \text{otherwise} \end{cases} \right) \right]$$

In conjunctive guarded systems, the initial states of all templates must be included in each guard [35].

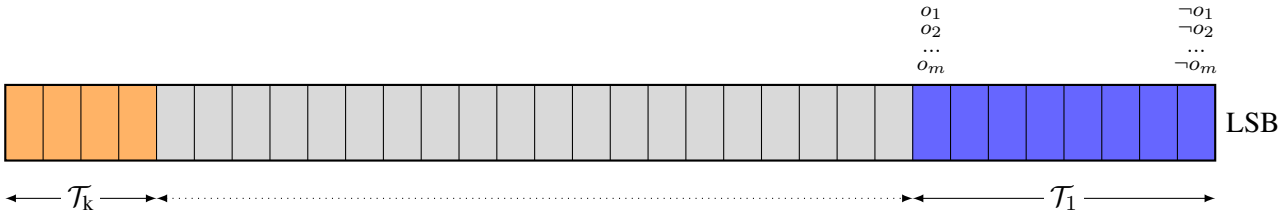


Figure 7.3: Guard bit vector partitioning

We ensure this by introducing the following constraint regarding the guard assignment function.

$$\bigwedge_{t,t' \in T_k} \bigwedge_{e \in \mathbb{B}^{E_k}} \big(\text{guard}(t,e,t') \neq 0\big) \to \big((\text{guard}(t,e,t') \,\&\, \text{guard\_initial}) = \text{guard\_initial}\big)$$

where $\text{guard\_initial} := \text{guard\_bit}_1(\text{init}_1) \,|\, \ldots \,|\, \text{guard\_bit}_k(\text{init}_k)$.

For checking whether the guard of some transition of process $\mathcal{T}_k^i$ is satisfied with respect to the current global state, the global state $s = (t_1^1, \ldots, t_1^{n_1}, t_2^1, \ldots t_{\mathrm{k}}^{n_k})$ is converted into a guard bit vector by considering all local states except for the local state of $\mathcal{T}_k^i$.

$$\begin{aligned}\text{guard\_set}_k^i(s) :=& \text{guard\_bit}_1(t_1^1) \mid \ldots \mid \text{guard\_bit}_1(t_1^{n_1}) \mid \text{guard\_bit}_2(t_2^1) \mid \ldots \mid \text{guard\_bit}_k(t_k^{i-1}) \mid \\ & \text{guard\_bit}_k(t_k^{i+1}) \mid \text{guard\_bit}_{\mathrm{k}}(t_{\mathrm{k}}^{n_k})\end{aligned}$$

Note that the guard_set function is only introduced for notational brevity, however, is not present in the final formula. Instead, the sequence of bit-wise disjunctions is embedded where the global state guard is used. Depending on the guard type, we define the guard evaluation function eval_guard, which takes two bit vectors with the same size. The first argument g_1 is the guard of some transition belonging to process $\mathcal{T}_k^i$ obtained by the function guard_k. The second guard g_2 is the current global state's guard representation for the particular process. For disjunctive guarded systems, we define

$$\text{eval\_guard}(g_1, g_2) := ((g_1 \& g_2) \neq 0)$$

The evaluation function for conjunctive guarded systems is

$$\text{eval\_guard}(g_1, g_2) := ((g_1 \mid g_2) = g_1) \wedge (g_1 \neq 0)$$

Based on the guard evaluation function, we define two additional template-specific wrapper functions. The first function $\text{delta\_enabled}_k : T_k \times \mathbb{B}^{|E_k|} \times T_k \times \mathbb{B}^G \to \mathbb{B}$ is true for some process $\mathcal{T}_k^i$ if the global state's guard representation $g_s := \text{guard\_set}_k^i(s)$ and the environment input e are such that transition $t \to t'$ is enabled.

$$\text{delta\_enabled}_k(t, e, t', g_s) := \text{eval\_guard}(\text{guard}_k(t, e, t'), g_s)$$

The second function $\text{any\_enabled}_k : T_k \times \mathbb{B}^{|E_k|} \times \mathbb{B}^G \to \mathbb{B}$ determines for process $\mathcal{T}_k^i$ in some current local state t whether there is an enabled transition with respect to the current environment input and the current global state.

$$\text{any\_enabled}_k(t, e, g_s) := \exists t' \in T_k : \text{delta\_enabled}(t, e, t', g_s)$$

Specification Encoding

In order to encode the specification, each property of the instantiated specification is first converted into a corresponding UCT. Then, the existence of a system that satisfies the UCT is encoded as finding a valid annotation for the run graph, as proposed by Finkbeiner and Schewe [69]. Here, each UCT is considered separately [52]. That is, for each UCT, we consider a system of the corresponding cutoff size. For each such subsystem, the environment provides a) inputs for each process (as defined by the particular process templates), and b) Boolean scheduling signals, which allow to schedule a single process instance. Valid process templates are found if each subsystem satisfies the encoded UCT. The advantage of this modular approach is that for each property we look for systems with cutoff size instead of systems whose size is the maximum of all cutoffs. This modular approach allows for a reduced size of the run graph compared to the run graph of the maximum cutoff sized system, because there are less states to be annotated. For example, in case of conjunctive guards, property cutoffs are constant, whereas deadlock detection cutoffs depend on the bound on the size of the particular process template.

The procedure of encoding a single UCT is illustrated in Algorithm 3. For representing the annotation of the run graph, we define two uninterpreted functions $\lambda^{\mathbb{B}}$ and $\lambda^{\#}$, where the first function determines whether a certain run graph state is annotated, and the latter assigns a concrete annotation value to a particular node. Here, each run graph state consists of a UCT state and a global state of the system with c_p processes (c_1 instances of template $\mathcal{T}_1$, c_2 instances of template $\mathcal{T}_2$ etc.). If $\lambda^{\mathbb{B}}$ is false for some node,

the node is annotated with the empty symbol, otherwise the annotation value is provided by $\lambda^{\#}$. Next, we ensure that the initial node of the run graph is annotated. In Line 5, the function get_schedulings is called in order to obtain the scheduling variable assignments for all possible schedulings in the system of size c_p.

Then, each UCT edge $q \xrightarrow{\text{lbl}} q'$ is iteratively analyzed. Because of the fairness constraints and inputs, which are only readable if the particular process moves, UCT labels possibly contain variables moves_k^i for different processes $\mathcal{T}_k^i$. Obviously, moves_k^i is false if the scheduling variables are such that a particular process $\mathcal{T}_k^i$ is not scheduled. In Line 9 of the algorithm, we therefore ensure that we only add SMT constraints for combinations of UCT edges and schedulings for which label and scheduling do not contradict each other. If the UCT label evaluates to false under the current scheduling, the considered UCT edge is not possible with respect to this scheduling. Otherwise, we add a run graph annotation constraint for the particular UCT edge $q \to q'$ and each combination of a) current global state, b) environment inputs for each process, and c) next global state with respect to the interleaving constraint, that is, only the currently scheduled process $\mathcal{T}_k^i$ changes its local state. For each such combination the run graph for q' and the next global system state must be annotated if

a) the run graph node for q and the current global system state is annotated,

b) inputs, outputs of processes, and architecture specific variables are such that the UCT's edge label is satisfied (represented by predicate cond), and

c) there is an enabled local transition from t_k^i to $t_k^{i\prime}$ with respect to the current environment inputs and the current global state guard.

The annotation value for the succeeding run graph node must be greater than the current run graph node, and strictly greater if the UCT's target node is rejecting. The conjunction cond (Item b) consists of one conjunct for each requirement defined by the UCT edge label. In order to represent our system model, we need to ensure that each input can only be read if the corresponding process moves. To this end, we encode each input requirement in cond as a conjunction of the particular quantified input variable and moves_k^i, where $\mathcal{T}_k^i$ is the process the input signal belongs to. Output requirements are encoded using the particular template's output functions. Architecture-specific variables are encoded in SMT as follows. is_enabled_k^i is replaced by the previously defined propositional function with the same name, i.e., $\text{is\_enabled}_k(t_k^i, e_k^i, g_s)$. is_init_k^i is replaced by a predicate $\text{is\_init}_k^i(t_k^i)$ which is true iff the quantified local state t_k^i is the initial state init_k. moves_k^i is encoded as $\text{is\_sched}_k^i \wedge \text{is\_enabled}_k^i$, with is_enabled_k^i being replaced by the predicate as described. Note that is_sched_k^i is an actual Boolean value determined by the encoder: true if the currently considered scheduling is such that process $\mathcal{T}_k^i$ is scheduled (without consideration of its enabledness), false otherwise.

For example, consider the UCT (Figure 7.4) that corresponds to the following instantiated liveness property.

$$\left(\mathsf{GF}\,\text{moves}_1^1 \wedge \mathsf{GF}\,\text{moves}_1^2 \wedge \mathsf{GF}\,\text{moves}_2^1\right) \to \left(\mathsf{G}(r_1^1 \wedge \text{moves}_1^1) \to \mathsf{F}\,g_1^1\right)$$

The environment provides two Boolean scheduling variables sched_0 and sched_1 to the property cutoff system of size $(1, 2)$. By setting both scheduling signals to false, the environment chooses to schedule process $\mathcal{T}_1^1$ (which causes is_sched_1^1 to become true). If sched_0 is true and sched_1 is false, process $\mathcal{T}_1^2$ is scheduled. Setting sched_0 to false and sched_1 to true causes process $\mathcal{T}_2^1$ to be scheduled. The assignment $\text{sched}_0 = \text{sched}_1 = \text{true}$ is invalid, since there is no fourth process in the system.

Consider the UCT edge $q_0 \to q_1$ with label $\text{moves}_1^1 \wedge \neg g_1^1 \wedge r_1^1$. By Line 9 of Algorithm 3, we only consider the scheduling $\text{sched}_0 = \text{sched}_1 = \text{false}$ for the encoding of this edge. Under consideration of this scheduling, we add a constraint in Line 16. The current global state consists of the three local states t_1^1, t_1^2, t_2^1. The next global state consists of a new local state $t_1^{1\prime}$ for the scheduled process as well as the current local states t_1^2 and t_2^1 of the other processes. Moreover, there is a single input variable, namely r_1^1, for which a universally quantified Boolean SMT variable is introduced.

Input: Formula f, UCT automaton $\mathcal{U}$, property cut-cutoff $c_p = (c_1, \ldots, c_\mathrm{k})$, global cutoff c
Output: SMT encoding of UCT automaton

$\lambda^{\mathbb{B}} := Q \times T_1 \times \cdots \times T_1 \times T_2 \times \ldots \times T_\mathrm{k} \to \mathbb{B}$
$\lambda^{\#} := Q \times T_1 \times \cdots \times T_1 \times T_2 \times \ldots \times T_\mathrm{k} \to \mathbb{N}$
$f \leftarrow \lambda^{\mathbb{B}}(q_0, \mathrm{init}_1, \ldots, \mathrm{init}_1, \mathrm{init}_2, \ldots, \mathrm{init}_\mathrm{k}) \wedge \lambda^{\#}(q_0, \mathrm{init}_1, \ldots, \mathrm{init}_1, \mathrm{init}_2, \ldots, \mathrm{init}_\mathrm{k}) = 0$

schedulings $\leftarrow$ get_schedulings(c_p, c)
foreach *UCT edge* $q \xrightarrow{lbl} q'$ **do**
  foreach *scheduling sched* $\in$ *schedulings* **do**
    $k, i \leftarrow$ get_scheduled_process$(scheduling)$
    if *scheduling* $\wedge$ *lbl* $==$ true **then**
      cond $\leftarrow$ build_conditions(lbl)
      $\triangleright \leftarrow \geq$
      if q' *is rejecting* **then**
        $\triangleright \leftarrow >$
      end

      $f \leftarrow f \wedge$
        $\Big[\forall t_1^1, \ldots, t_1^{c_1}, t_2^1, \ldots, t_\mathrm{k}^{c_\mathrm{k}}\, \forall t_k^{i\prime}\, \forall e_1^1, \ldots, e_1^{c_1}, \ldots, e_\mathrm{k}^{c_\mathrm{k}}$:
        $\Big(\lambda^{\mathbb{B}}(q, t_1^1, \ldots, t_\mathrm{k}^{c_\mathrm{k}}) \wedge \mathrm{cond} \wedge \mathrm{delta\_enabled}(t_k^i, e_k^i, t_k^{i\prime}, \mathrm{guard\_set}_k((t_1^1, \ldots, t_\mathrm{k}^{c_\mathrm{k}})), i)\Big) \rightarrow$
        $\Big(\lambda^{\mathbb{B}}(q', t_1^1, \ldots, t_k^{i\prime}, \ldots t_\mathrm{k}^{c_\mathrm{k}}) \wedge \lambda^{\#}(q', t_1^1, \ldots, t_k^{i\prime}, \ldots t_\mathrm{k}^{c_\mathrm{k}}) \triangleright \lambda^{\#}(q, t_1^1, \ldots, t_k^i, \ldots t_\mathrm{k}^{c_\mathrm{k}})\Big)\Big]$

    end
  end
end
return f

Algorithm 3: UCT Encoding Algorithm

cond is built based on the label. The first conjunct of the label, moves_1^1, is replaced by $\mathrm{is\_sched}_1^1 \wedge \mathrm{is\_enabled}_1(t_1^1, r_1^1, \mathrm{guard\_bit}_1(t_1^2) \mid \mathrm{guard\_bit}_2(t_2^1))$, where $\mathrm{is\_sched}_1^1$ is obviously true. The second conjunct, namely $\neg g_1^1$, contains an output signal of template $\mathcal{T}_1$, for which there exists an SMT function $g_1 : T_1 \to \mathbb{B}$. Thus, we represent $\neg g_1^1$ as $\neg g_1(t_1^1)$. The last conjunct contains the input signal r_1^1. This signal corresponds to the universally quantified SMT variable with the same name. As described above, we encode this signal as $r_1^1 \wedge \mathrm{moves}_1^1 = r_1^1 \wedge \mathrm{true} \wedge \mathrm{is\_enabled}_1(t_1^1, r_1^1, \mathrm{guard\_bit}_1(t_1^2) \mid \mathrm{guard\_bit}_2(t_2^1))$.

The resulting constraint is as follows.

$$\forall t_1^1, t_1^2, t_1^{1\prime} \in T_1\, \forall t_2^1 \in T_2\, \forall r_1^1 \in \mathbb{B} :$$
$$\Big(\lambda^{\mathbb{B}}(q_0, t_1^1, t_1^2, t_2^1) \wedge \mathrm{is\_enabled}_1(t_1^1, r_1^1, \mathrm{guard\_bit}_1(t_1^2) \mid \mathrm{guard\_bit}_2(t_2^1)) \wedge \neg g_1(t_1^1) \wedge r_1^1 \wedge$$
$$\mathrm{delta\_enabled}(t_1^1, r_1^1, t_1^{1\prime}, \mathrm{guard\_bit}_1(t_1^2) \mid \mathrm{guard\_bit}_2(t_2^1)))\Big) \rightarrow$$
$$\Big(\lambda^{\mathbb{B}}(q_1, t_1^{1\prime}, t_1^2, t_2^1) \wedge \lambda^{\#}(q_1, t_1^{1\prime}, t_1^2, t_2^1) > \lambda^{\#}(q_0, t_1^1, t_1^2, t_2^1)\Big)$$

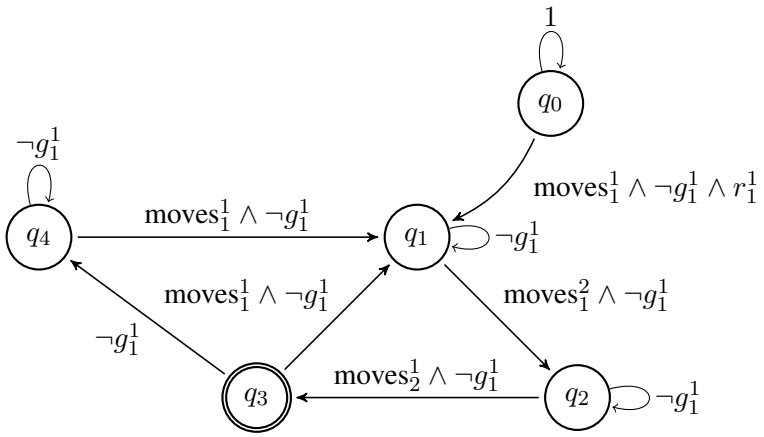


Figure 7.4: UCT of an illustrative liveness property

7.2 Remarks and Optimizations

In this section, we consider optimizations of the encoding based on observations about the relation between state labels and actual states as well as additional heuristics and optimizations mentioned in previous work on bounded synthesis.

7.2.1 Label-Based Guards

In the original system model by Emerson and Kahlon [35], guards are defined over states instead of output labels. In our system model (see Chapter 4), there are two possible cases which need to be further investigated. First, if the number of observable states $|\mathbb{B}^{O_k}|$ is smaller than the number of states $|T_k|$ for some template $\mathcal{T}_k$, we are not able to uniquely identify each state $t \in T_k$. Instead, the internal system states T_k are partitioned, because multiple states are assigned to the same label, and thus belong to the same observable state. In the second case, $|\mathbb{B}^{O_k}| > |T_k|$. Here, it depends on the concrete implementation whether states are uniquely identifiable (each state is assigned a different label), or whether there are multiple states that appear as a single observable state (at least one label is assigned to more than one state).

Consider a system where $|\mathbb{B}^{O_l}| = |T_l|$ for all templates $\mathcal{T}_l$ except for some template $\mathcal{T}_k$. Compared to the original system model from [35], using the concept of observable states yields a smaller problem instance if $|\mathbb{B}^{O_k}| < |T_k|$, since there exist less possible guards. However, there possibly exists a solution which requires that all states T_k are uniquely identifiable. Using observable states, this problem can be resolved by introducing *auxiliary labels*, that is, adding pseudo-variables to O_k such that the number of used labels exceeds $|T_k|$. The actual number of required auxiliary variables depends on state partitioning, i.e., the number of observable states. Under the assumption that the cardinality of the original output set O_k is at least 1 and the specification allows at least two output assignments (all output signals are true and false, respectively), the upper bound for the number of required auxiliary labels is $\log |T_k| - 1$.

Obviously, $|\mathbb{B}^{O_k}| > |T_k|$ yields a larger number of possible guards, therefore, the SMT problem instance becomes more difficult to solve. One way to ensure that $|\mathbb{B}^{O_k}| \leq |T_k|$ is to split the set of labels O_k into two sets: a set of labels that are considered as part of the observable state, and a set of labels whose assignment does not influence any guard. However, this partitioning is not known a priori, since it depends on the semantics of the different output labels defined by the specification. Possible solutions are discussed in the Section 7.2.2. Note that if $|T_k|$ is not a power of 2, adding all auxiliary variables required to uniquely identify each state causes the number of possible label combinations (observable states) to be larger than $|T_k|$. Solutions to this problem are discussed in Section 7.2.3.

7.2.2 State-Based Guards

As already pointed out, using label-based guards instead of state-based guards is more complex if $\sum_k |\mathbb{B}^{O_k}| > \sum_k |T_k|$. To this end, we distinguish between two cases when defining the architecture-specific constraints for each template. On the one hand, if $\sum_k |\mathbb{B}^{O_k}| \leq \sum_k |T_k|$, we use label-based guards as defined in our system model. On the other hand, if $\sum_k |\mathbb{B}^{O_k}| > \sum_k |T_k|$, we use state-based guards for the particular template and define guard_bit_k to map each state $t \in T_k$ to a unique bit inside the bit chunk assigned to template $\mathcal{T}_k$. The following function determines the number of bits in the guard bit vector allocated to a particular template $\mathcal{T}_k$.

$$\text{chunk\_size}(\mathcal{T}_k) := \min(|\mathbb{B}^{O_k}|, |T_k|)$$

Then, the guard bit vector size is equal to $\sum_k \text{chunk\_size}(\mathcal{T}_k)$. Let the function $\text{state\_index}_k : T_k \to [0, |T_k| - 1]$ map each state a to a unique number. The guard bit function distinguishes between two cases as follows.[3]

$$\text{guard\_bit}_k(t) := 1 \ll \left[\left(\sum_{l=0}^{k-1} \text{chunk\_size}(\mathcal{T}_l) \right) + \begin{cases} \text{label\_bit\_pos}(t) & \text{if chunk\_size}(\mathcal{T}_k) = |\mathbb{B}^{O_k}| \\ \text{state\_bit\_pos}(t) & \text{otherwise} \end{cases} \right]$$

$$\text{label\_bit\_pos}(t) := \sum_{i=1}^{|O_k|} \left(\begin{cases} 1 \ll (i-1) & \text{if } o_{k_i}(t) \\ 0 & \text{otherwise} \end{cases} \right)$$

$$\text{state\_bit\_pos}(t) := \text{state\_index}_k(t)$$

7.2.3 Auxiliary Variables

As already mentioned, label-based guards allow to partition the actual state space into observable states, and thus provide support for making states *private*. However, if the number of observable states is smaller than the number of states, bounded synthesis possibly fails to find a solution because the number of available guard variables is too small. Auxiliary variables increase the number of possible output assignments, and therefore can be used to achieve a more fine-grained partitioning of states into observable states.

In the following, we present an extension for Algorithm 2, which is based on label-based guards. By iteratively adding auxiliary variables, it increases the number of available output assignments the SMT solver can choose, and therefore ensures that we do not miss solutions because of a too coarse state partitioning (and consequently too few guard variables). In contrast to immediately adding the maximum number of auxiliary variables to O_k, the iterative approach ensures that the returned solution is minimal with respect to the required number of auxiliary variables, and thus has the coarsest state partitioning. Algorithm 4 shows the modifications to Algorithm 2, beginning in Line 10. Our optimization requires an additional loop inside the original main loop. In Line 15, we define an initially empty set of auxiliary labels for each template. We then try to find a solution for the bounded synthesis problem as in the original algorithm. However, if the solver's response is UNSAT, we do not increase the bound immediately. Instead, we add to each set aux_k an additional auxiliary variable unless the maximum number of required auxiliary variables is reached. New auxiliary variables are also added to the output set O_k of the particular process template. Then, the inner while loop is repeated if at least one new auxiliary variable was added. This inner loop is executed either until a solution is found, or each output set O_k contains enough auxiliary variables to uniquely identify each state. In the latter case, we increase the template bounds, and repeat the outer while loop defined in Line 2. This extension can also be refined to additionally include a combination of the state-based and label-based approach, as described in Section 7.2.2. One possible solution is to decide for each template, whether the number of label-based guard variables

[3] Note that $o_{k_i}(t)$ denotes the i-th component of the output label for template $\mathcal{T}_k$ in state t.

$|\mathcal{B}^{O_k \cup \text{aux}_k}|$ is smaller than the number of state-based guard variables $|T_k|$. In the positive case, the algorithm encodes the template using label-based guards, in the negative case using state-based guards. This refinement unites the advantages of both label-based and state-based guards, and keeps the number of guard variables (and thus the problem size) is kept as small as possible.

```
9  ...
10 while increments < max _increments do
11     C_p, ← get_cutoffs(Φ′, N, b)
12     all_state_based ← false
13     φ ← instantiate(Φ′, b, C_p)
14     foreach k ∈ [1, k] do
15         aux_k ← ∅
16     end
17     while ¬all_state_based do
18         formula ← encode(φ, b, C_p)
19         res ← solve(formula)
20         if res == SAT then
21             return extract_model()
22         end
23         all_state_based ← true
24         foreach k ∈ [1, k] do
25             if |B^|aux_k|+1| < |T_k| then
26                 aux_k ← aux_k ∪ {aux_k^|aux_k|}
27                 O_k ← O_k ∪ aux_k
28                 all_state_based ← false
29             end
30         end
31     end
32     b ← (b_1 + 1, ..., b_k + 1)
33     increments ← increments + 1
34 end
35 return UNSAT
```

Algorithm 4: Optimized Bounded Synthesis of Guarded Systems

7.2.4 Further Optimizations

We observe monotonicity for both conjunctive and disjunctive guards. If a guard g is satisfied by some global state guard g_s, then g_s also satisfies each guard g' which is such that $g \,\&\, g' = g$, i.e. all guard variables that are true in g are also true in g'. Although monotonicity decreases the number of different guards the solver needs to consider until an appropriate one is found, we can further restrict the set of guards by introducing heuristics, and preferring the search for guards with a specific structure. Disjunctive guards are satisfied if at least one process is in some specific state. From this definition we infer that searching for guards with a small cardinality (i.e., a small number of guard variables is true) is a reasonable heuristic. By contrast, conjunctive guards are disabled by a process in some specific state. As a consequence, preferring conjunctive guards with a large number of true guard variables seems reasonable.[4] This optimization is realizable in a similar way as described for iteratively added auxiliary variables.

[4] This is enforced by our theoretical results for conjunctive guarded systems, which are currently restricted to 1-guards.

Finkbeiner and Schewe [38] describe a set of optimizations, some of which we implicitly used in our description of the encoding (e.g., input elimination). Another encoding-specific optimization mentioned in [38] as well as by Khalimov et al. [52] is to reduce the number of run graph states that need to be annotated by exploiting the structure of the particular encoded UCT. In this optimization, only UCT states that are part of a rejecting SCC (an SCC with at least one rejecting state) need to be considered by the labelling functions.

Chapter 8

Implementation and Experiments

8.1 Implementation

We developed a command-line prototype application that implements our guarded synthesis approach. To this end, we reused parts of the Python-based bounded synthesis tool PARTY [51]. Our prototype supports synthesis of templates with state-based or label-based guards, and is ready for using auxiliary variables, SCCs in run graphs, and other optimizations discussed in the previous chapter.

Figure 8.1 illustrates the main steps during the execution of our application. The tool takes an indexed LTL\X specification as well as properties of the desired system, including the guard type, the maximum bound for each process template, and the desired system size.[1] In the preprocessing step, each liveness property is weakened by adding an assumption expressing unconditional fairness. To this end, we determine the property types by analyzing the corresponding UCT as described in Chapter 7. We obtain the desired UCT by converting the negated property into a Büchi automaton. For this task, we use LTL3BA[2] [8], the successor of the LTL-to-Büchi conversion tool LTL2BA [40]. The outcome of LTL3BA is then interpreted as a UCT. After processing all specification properties, the indexed LTL\X specification is extended by architectural constraints (provided by System Class), and instantiated under consideration of the desired template size and the applicable cutoffs (specified by the System Class). In the succeeding step, the instantiated LTL formula is converted into a UCT automaton using LTL3BA. Then the UCT automata, architectural properties (in particular the semantics of process template and

[1] Note that our prototype application is intended to be used for evaluation purposes. At the moment, it does not support automatic computation of cutoffs, and the user must define the size of the system (i.e., the number of processes) to be synthesized.

[2] Available at `http://sourceforge.net/projects/ltl3ba/`.

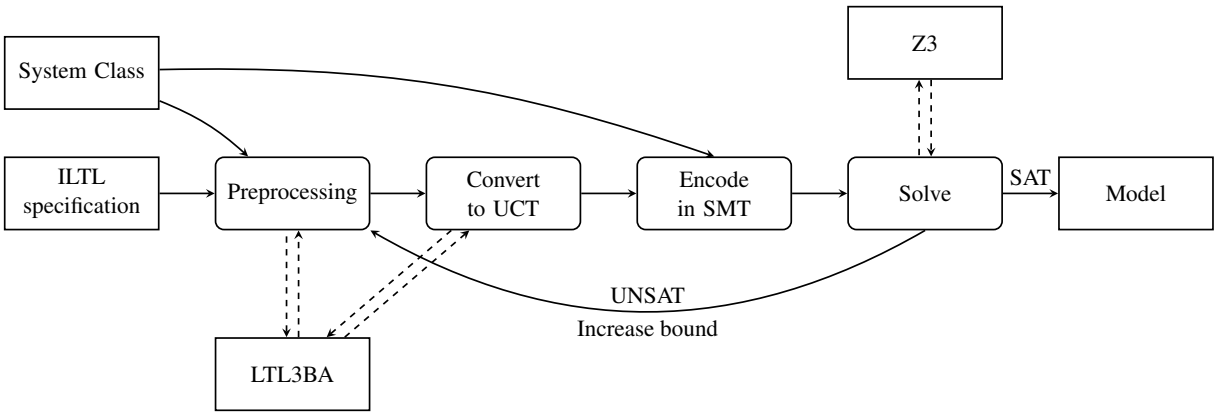


Figure 8.1: Prototype program flow

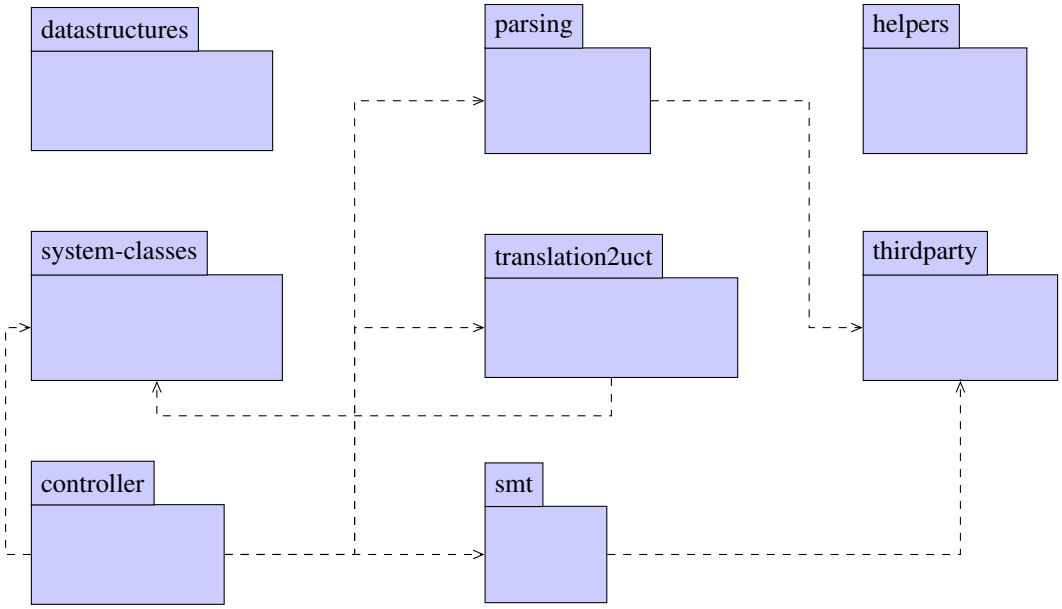


Figure 8.2: Prototype package diagram

guards) provided by System Class, and bounded synthesis constraints are encoded in SMT. The resulting SMT instance is then solved using the SMT solver Z3 [25]. Note that our prototype relies on the Z3 Python API [3] for encoding and solving SMT problems. If the SMT instance is satisfiable, we obtain process templates from the model returned by the SMT solver. Otherwise, the bounds for the process template sizes are increased, and all steps starting from the property instantiation are repeated. This loop is terminated if either a model is retrieved, or the current process template size exceeds the maximum bound defined by the user.

Figure 8.2 shows the program's components. In the following paragraphs, we describe each component (i.e., Python package, or Python module).

System Classes The package `system-classes` provides an extendable hierarchy of system class descriptions, defining fairness, architecture-specific and system-specific assumptions and guarantees, cutoff calculation routines, and various special restrictions that apply to the particular system class. So far, the tool supports conjunctive and disjunctive guarded systems.

Specification Parsing The specification file parsing functionality (package `parsing`) is reused from PARTY and extended in order to support quantified template variables. Template variables have the structure `<name>_k`, where `<name>` is some unique identifier consisting of letters and digits, and `k` denotes the index of the template the signal belongs to. Quantified template variables have the structure `<name>_k_i`, where `<name>` and `k` are as for template variables, and `i` is the index of the process instance (w.r.t. template `k`). An example specification is shown in Listing 8.1. It is structured into multiple sections. The first section `GENERAL` contains information about the number of templates used in the system. In the section `INPUT_VARIABLES` input variables for each template are defined (given as template variables). Likewise, the section `OUTPUT_VARIABLES` lists output variables for each template. In our example, there is a single template with an input signal `r_0` and an output signal `g_0`. The section `LABEL_VARIABLES` contains information that is considered by the label-based approach only. It defines all output variables which are part of the observable state. The section `ASSUMPTIONS` contains assumptions under which the succeeding guarantees must hold (currently not used because the supported properties described in Section 4.3 only consist of guarantees). The section `GUARANTEES`

[3] Available at `https://github.com/z3prover`.

includes guarantees that must be satisfied by the desired system implementation. Our example lists a single guarantee, namely mutual exclusion.

```
1  [GENERAL]
2  templates: 1
3
4  [INPUT_VARIABLES]
5  r_0;
6
7  [OUTPUT_VARIABLES]
8  g_0;
9
10 [LABEL_VARIABLES]
11 g_0;
12
13 [ASSUMPTIONS]
14
15 [GUARANTEES]
16 Forall (i,j) G(!(g_0_i=1 * g_0_j=1));
```

Listing 8.1: Example for a well-formed specification

UCT Translation The UCT translation functionality is reused from PARTY. As mentioned above, our tool does not implement the translation routine, but instead relies on LTL3BA [40]. Our implementation is thus responsible for calling LTL3BA, providing the appropriate input, and parsing the external application's output.

SMT Encoding The package `smt` provides functionality for encoding the bounded synthesis problem and solving the SMT instance. Moreover, this component is responsible for extracting information about process implementations from a model obtained by the SMT solver.

Other components The module `controller` is responsible for coordinating the synthesis process and implements the loop illustrated in Figure 8.1. The packages `datastructures`, `helpers` provide common functionality used by various components. The package `thirdparty` encapsulates third-party components (parser, Z3 bindings etc.).

8.2 Experiments

The evaluation focuses on the impact of our cutoffs on the runtime. We aim at observing whether the particular cutoffs are low enough to allow us to synthesize a solution in a reasonable time. Furthermore, we are interested in identifying the main contributors in case of high runtimes. To this end, we consider different specifications for (conjunctive and disjunctive) guarded systems. Note that in a real-world scenario, we would solve the synthesis problem for all system sizes up to the cutoff system in a single run in order to obtain a parameterized system. However, for evaluation purposes we sequentially find solutions for an increasing number of instances. We also compare the runtime of the label-based approach with the runtime of the state-based approach, where applicable, i.e., where the number of states of at least one template is not equal to the number of possible label assignments (observable states). For our experiments we use a PC with an Intel i7-3770K processor and 16GB RAM. For performance reasons we do not use any swapping mechanism. The run time is determined by averaging over 10 runs. We set the timeout period to 1.5 hours.

8.2.1 Conjunctive Guarded Systems

Example 1

Example 1 is a simple arbiter, consisting of one process template. It takes an input `r` and provides an output `g`. The specification requires that each request `r` is finally granted, and that there are no two grants at the same time. For label-based synthesis, the observable state depends on the grant signal.

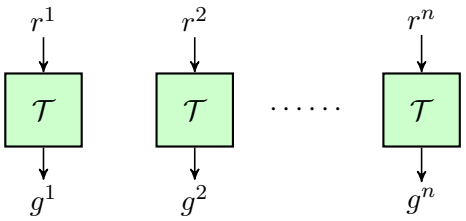


Figure 8.3: Conjunctive Example 1: Black box view

```
1  [GENERAL]
2  templates: 1
3
4  [INPUT_VARIABLES]
5  r_0;
6
7  [OUTPUT_VARIABLES]
8  g_0;
9
10 [LABEL_VARIABLES]
11 g_0;
12
13 [GUARANTEES]
14 Forall (i) G(r_0_i=1 -> F(g_0_i=1));
15 Forall (i,j) G(!(g_0_i=1 * g_0_j=1));
```

Listing 8.2: Conjunctive Example 1: Specification

The synthesized process template is shown in Figure 8.5. It consists of two states — the non-granting initial state t^0, and the granting state t^1. A process only moves if every other process in its initial state. Raising grant signals is not necessarily preceded by requesting the resource, however, each request

Instances	Runtime state-based		Runtime label-based	
	with cutoffs	without cutoffs	with cutoffs	without cutoffs
2	0.93	0.93	0.93	0.93
3	1.1	5.5	1.1	5.5
4	1.1	42.9	1.1	43.5
5	1.1	631.1	1.1	641.1
6	1.1	TIMEOUT	1.1	TIMEOUT

Table 8.1: Conjunctive Example 1: Runtimes

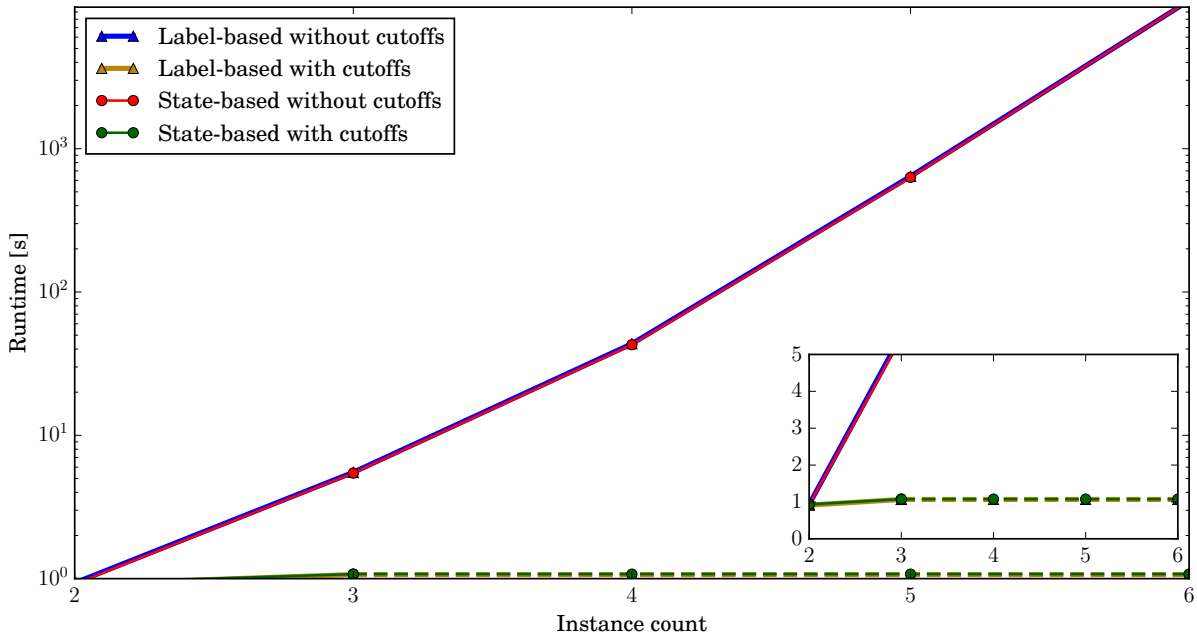


Figure 8.4: Conjunctive Example 1: Runtime

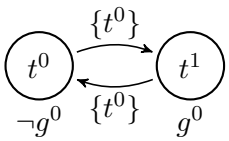


Figure 8.5: Conjunctive Example 1: Process template

is finally granted, and thus the specification is satisfied. From Table 8.1 we can see that the runtime increases exponentially if no cutoffs are used, a distributed system with 6 processes cannot be synthesized within the TIMEOUT period. For the given specification, and $|\mathcal{T}| = 2$, the deadlock detection cutoff and the property cutoff for the single-indexed property is (2), the cutoff for the double-indexed property is (3). For this reason, the runtime for synthesis with cutoffs remains constant with respect to the number of processes for $n \geq 3$. Because the number of possible output assignments is equal to the number of template states, we cannot observe any significant runtime differences between state- and label-based synthesis.

Example 2

Example 2 is an extended version of Example 1. Additional specification properties ensure the absence of both initial spurious grants and spurious grants after the grant state is visited.

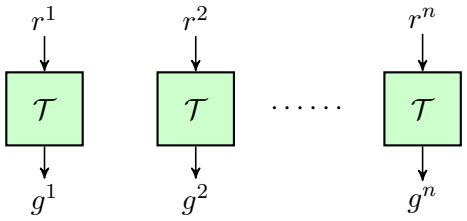


Figure 8.6: Conjunctive Example 2: Black box view

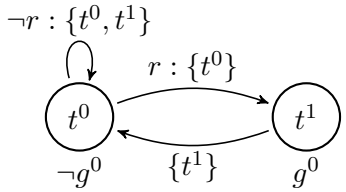


Figure 8.7: Conjunctive Example 2: Process template

```
1  [GENERAL]
2  templates: 1
3
4  [INPUT_VARIABLES]
5  r_0;
6
7  [OUTPUT_VARIABLES]
8  g_0;
9
10 [LABEL_VARIABLES]
11 g_0;
12
13 [GUARANTEES]
14 Forall (i) G(r_0_i=1 * -> F(g_0_i=1));
15 Forall (i,j) G(!(g_0_i=1 * g_0_j=1));
16 Forall (i) (!((r_0_i=0 * g_0_i=0) U (r_0_i=0 * g_0_i=1)));
17 Forall (i) G(g_0_i=1 -> (g_0_i=1 U ((g_0_i=0 U r_0_i=1) + G(g_0_i=0))));
```

Listing 8.3: Conjunctive Example 2: Specification

The implementation of the synthesized process template (Figure 8.7) is similar to the implementation of Example 1. Spurious grants are avoided by only granting if a request is seen. Note that the imple-

Instances	Runtime state-based		Runtime label-based	
	with cutoffs	without cutoffs	with cutoffs	without cutoffs
2	1.9	1.9	1.8	1.8
3	2.0	10.3	2.0	10.1
4	2.0	72.0	2.0	71.4
5	2.0	1054	2.0	1046
6	2.0	TIMEOUT	2.0	TIMEOUT

Table 8.2: Conjunctive Example 2: Runtimes

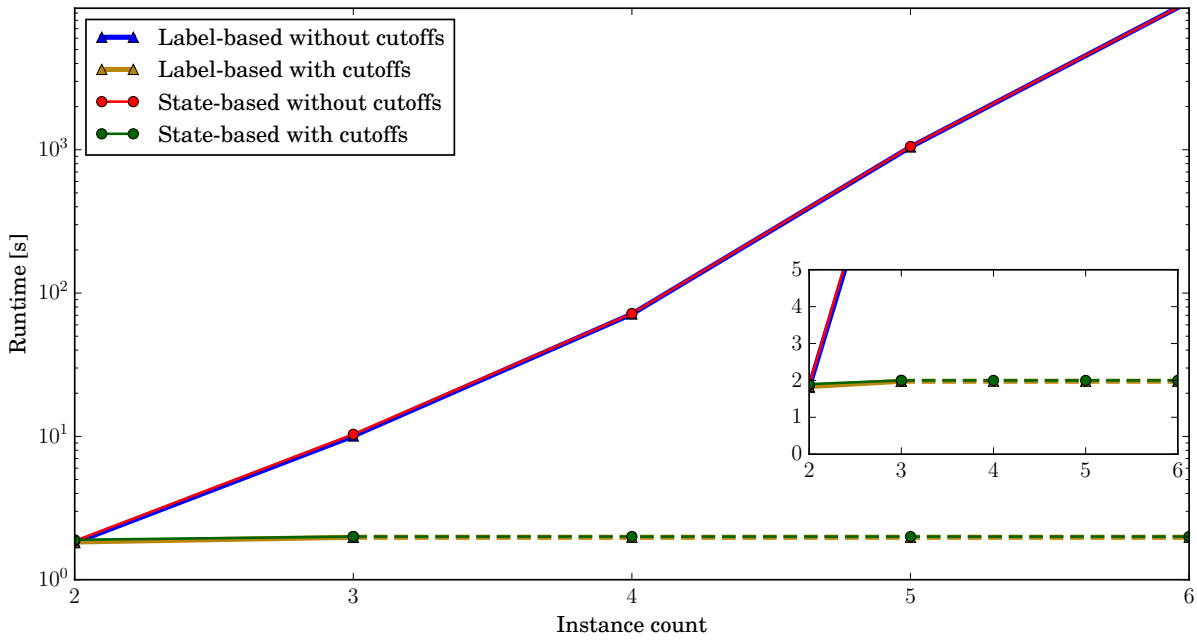


Figure 8.8: Conjunctive Example 2: Runtime

mentation does not react to requests it cannot see, that is, a) requests raised when the process is not scheduled, and b) requests which are raised when the process is scheduled, but cannot move. This behaviour is allowed by our system model as long as fairness is ensured. Table 8.1 shows that the runtimes for Example 2 are similar to the ones for Example 1. Overall, we observe slightly higher runtimes due to the additional properties. In this example, the cutoffs are also (2) and (3) respectively. Therefore, the runtime remains constant for systems with more than two processes. Because the number of states is equal to the number of output variable assignments, there is no significant difference between the state-based and the label-based encoding.

Example 3

Example 3 consists of one template with two requests $r1$, $r2$, and two grants $g1$, $g2$. Both requests must be granted eventually, however, request $r2$ has lower priority than $r1$. That is, if both resources are requested, only the first must be eventually granted. Requests during a grant phase can be ignored. Mutual exclusion is specified resource-wise and for pairs of different grants. Moreover, spurious grants must be avoided. For the label-based approach, we define both grant signals to be part of the observable state.

```
1  [GENERAL]
2  templates: 1
3
4  [INPUT_VARIABLES]
5  r1_0;
6  r2_0;
7
8  [OUTPUT_VARIABLES]
9  g1_0;
10 g2_0;
11
12 [GUARANTEES]
13 Forall (i,j) G(!(g1_0_i=1 * g1_0_j=1));
14 Forall (i,j) G(!(g2_0_i=1 * g2_0_j=1));
15 Forall (i) G(!(g1_0_i=1 * g2_0_i=1));
16
17 Forall (i) G((r1_0_i=1 * g2_0_i=0) -> (F(g1_0_i=1)));
18 Forall (i) G((r2_0_i=1 * g1_0_i=0 * r1_0_i=0) -> (F(g2_0_i=1)));
19
20 Forall (i) G(g1_0_i=1 ->
21              (g1_0_i=1 U ((g1_0_i=0 U r1_0_i=1) + G(g1_0_i=0))));
22 Forall (i) G(g2_0_i=1 ->
23              (g2_0_i=1 U ((g2_0_i=0 U r2_0_i=1) + G(g2_0_i=0))));
24
25 Forall (i) (!((r1_0_i=0 * g1_0_i=0) U (r1_0_i=0 * g1_0_i=1)));
26 Forall (i) (!((r2_0_i=0 * g2_0_i=0) U (r2_0_i=0 * g2_0_i=1)));
```

Listing 8.4: Conjunctive Example 3: Specification

As shown in Figure 8.11, there exist different valid implementations because we do not specify how the system has to react in case of requests while being in one of the grant states. The implementation in Figure 8.11a ignores requests while granting one resource, whereas the implementation in Figure 8.11b grants requests for resource 1 in any case. From Table 8.3 we see that for $n = 2$, an implementation can be found although no cutoffs are used. The constant property cutoffs ($c = 2$ for single-indexed, and $c = 3$ for double-indexed properties) significantly reduce the complexity, and thus allow to synthesize systems with more than 2 processes. In case of the state-based encoding, the runtime remains constant

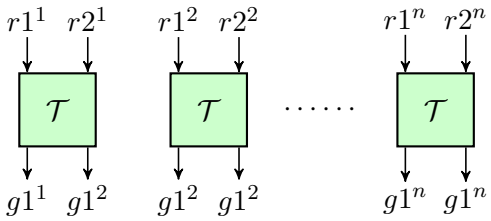


Figure 8.9: Conjunctive Example 3: Black box view

Instances	Runtime state-based		Runtime label-based	
	with cutoffs	without cutoffs	with cutoffs	without cutoffs
2	48.6	44.5	46.3	47.3
3	195.0	1946	2380	TIMEOUT
4	3001	TIMEOUT	TIMEOUT	-
5	3001	-	-	-
6	3001	-	-	-

Table 8.3: Conjunctive Example 3: Runtimes

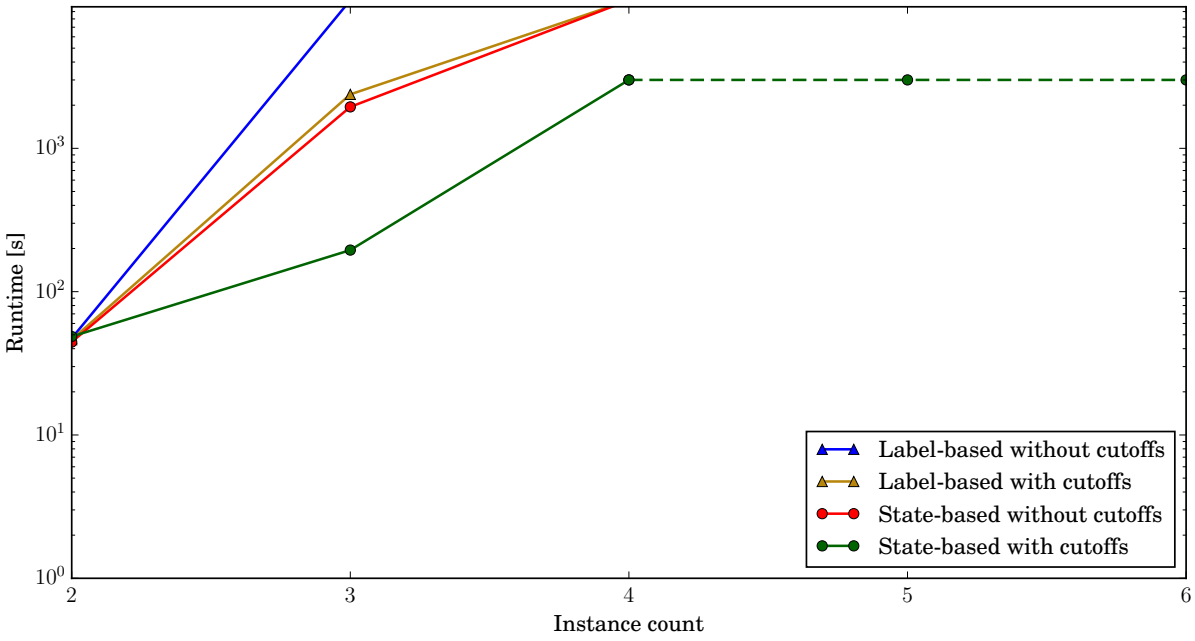
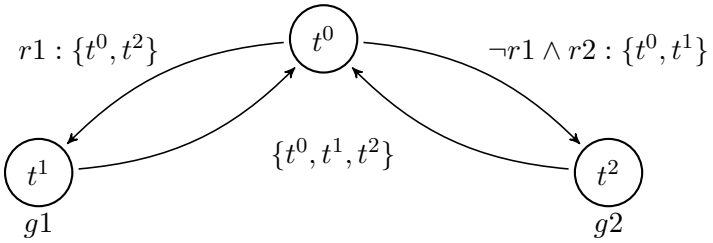
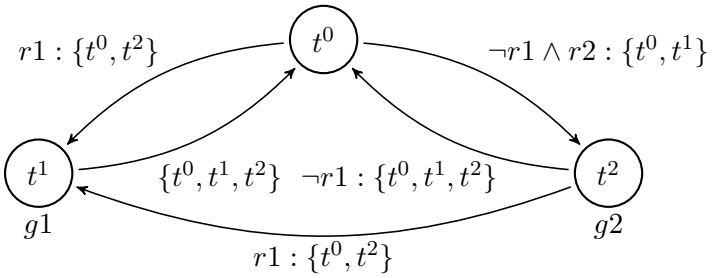


Figure 8.10: Conjunctive Example 3: Runtime



(a) Implementation not considering requests in grant states



(b) Implementation considering requests $r1$ in grant state t^2

Figure 8.11: Conjunctive Example 3: Process templates (not complete)

for $n \geq 4$ because of the deadlock detection cutoff ($c = 4$). For a bound $b < 4$, the state-based approach yields a smaller number of guard variables (cardinality b) than the label-based approach (4 possible assignments for output variables). Thus, we observe smaller runtimes for state-based synthesis. The increased complexity of label-based synthesis in this example even yields a timeout for $n \geq 4$.

8.2.2 Disjunctive Guarded Systems

Example 1

Example 1 specifies a disjunctive guarded system consisting of two process templates, with multiplicity vector $(1, n-1)$, that is, independent from the system size there is only one process belonging to the first template. Each template has a single output (w, g, resp.), and no environment inputs. In the desired system, instances of both templates must change the state of their outputs infinitely often, and infinitely often, there must be one process pair i, j, for which w^i and g^j are true at the same time. For label-based synthesis, all output signals are defined to be part of the observable state.

```
1  [GENERAL]
2  templates: 2
3
4  [INPUT_VARIABLES]
5
6  [OUTPUT_VARIABLES]
7  w_0;
8  g_1;
9
10 [GUARANTEES]
11 Forall (i) (G(F(w_0_i=0)) * G(F(w_0_i=1)));
12 Forall (i) (G(F(g_1_i=0)) * G(F(g_1_i=1)));
13 Forall (i,j) G(F(w_0_i=1 * g_1_j=1));
```

Listing 8.5: Disjunctive Example 1: Specification

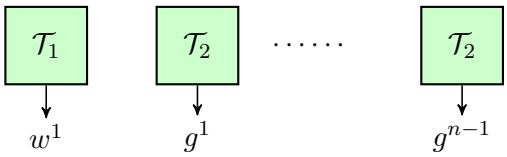


Figure 8.12: Disjunctive Example 1: Black box view

Instances	Runtime state-based		Runtime label-based	
	with cutoffs	without cutoffs	with cutoffs	without cutoffs
2	1.9	1.9	1.9	1.9
3	9.5	9.8	9.3	9.5
4	110.2	99.6	80.4	79.9
5	128.6	1496	92.9	1309
6	128.6	TIMEOUT	92.9	TIMEOUT

Table 8.4: Disjunctive Example 1: Runtimes

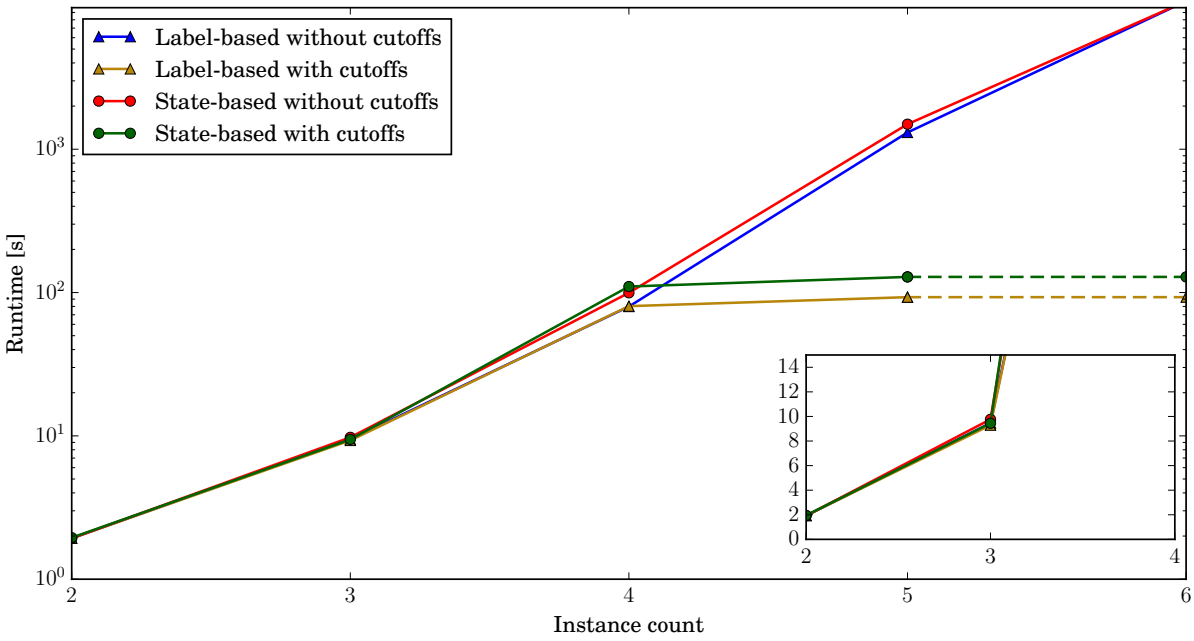


Figure 8.13: Disjunctive Example 1: Runtime

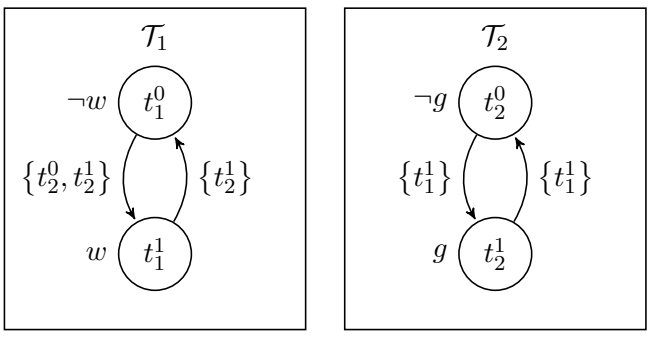


Figure 8.14: Disjunctive Example 1: Process template

The implementation is shown in Figure 8.14. Both process templates lower the output signals in their initial states (t_1^0, t_2^0). The guards of the two templates ensure that for all pairs of $\mathcal{T}_1$, $\mathcal{T}_2$ processes, w and g are raised infinitely often at the same time. The $\mathcal{T}_1$ process can always move to state t_1^1, but can only move back to the initial state if there is at least one $\mathcal{T}_2$ process in the grant state. Template $\mathcal{T}_2$ processes can only move if the $\mathcal{T}_1$ process is in state t_1^1. Table 8.4 lists the runtime results. We observe that applying cutoffs yields constant runtimes for $n \geq 5$. This is due to the single-indexed and double-indexed property cutoffs ($c = (4, 4)$, clipped to $c = (1, 4)$). Furthermore, the deadlock detection cutoff $c = (3, 3)$ (clipped to $c = (1, 3)$) causes the runtime improvement for $n = 4$. The relatively small runtime difference between $n = 4$ and $n = 5$ can be explained by the fact that in case of $n = 5$ the UCT automata for the specified properties have only up to 2 additional states compared to $n = 4$. This also suggests that the deadlock detection property is a major contributor to the runtime increase. The runtime difference between label-based encoding and state-based encoding has no theoretical reason, however, it suggests that the solver "profits" from the label-based encoding for this example.

Example 2

Example 2 is a more complex specification. It defines two templates with multiplicity vector $(1, n-1)$. The first template has one output read. The second template takes two inputs req and reqtype, and has two outputs send and sendtype. According to the specification, the process belonging to the first template must change its read output infinitely often. Processes belonging to the second template must react on each request req with a certain reqtype by eventually raising the send output with the corresponding sendtype. Furthermore, the first process must read when a send signal is raised. For label-based synthesis, we specify that the observable states of the processes only consider the signal read of the first process template, and the signal send of the second template.

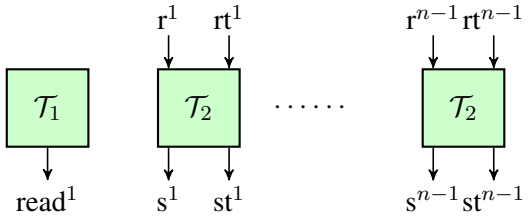


Figure 8.15: Disjunctive Example 2: Black box view

```
[GENERAL]
templates: 2

[INPUT_VARIABLES]
req_1;
reqtype_1;

[OUTPUT_VARIABLES]
read_0;
sendtype_1;
send_1;

[LABEL_VARIABLES]
read_0;
send_1;

[GUARANTEES]
Forall (i) (G(F(read_0_i=1)) * G(F(read_0_i=0)));
Forall(i,j) (G((req_1_i=1 * reqtype_1_i=0) ->
```

```
20                F(send_1_i=1 * sendtype_1_i=0 * read_0_j=1)));
21 Forall(i,j) (G((req_1_i=1 * reqtype_1_i=1) ->
22                F(send_1_i=1 * sendtype_1_i=1 * read_0_j=1)));
23
24 Forall (i) (!((req_1_i=0 * send_1_i=0) U (req_1_i=0 * send_1_i=1)));
25 Forall (i) G(send_1_i=1 ->
26              (send_1_i=1 U ((send_1_i=0 U req_1_i=1) + G(send_1_i=0))));
27
28 Forall(i) !(send_1_i=1 U req_1_i=1);
29 Forall(i) (G((send_1_i=1) -> (send_1_i=1 U (send_1_i=0 U req_1_i=1))));
```

Listing 8.6: Disjunctive Example 2: Specification

Instances	Runtime state-based		Runtime label-based	
	with cutoffs	without cutoffs	with cutoffs	without cutoffs
2	4.35	4.35	4.32	4.32
3	TIMEOUT	TIMEOUT	TIMEOUT (*)	TIMEOUT (*)

Table 8.5: Disjunctive Example 2: Runtimes

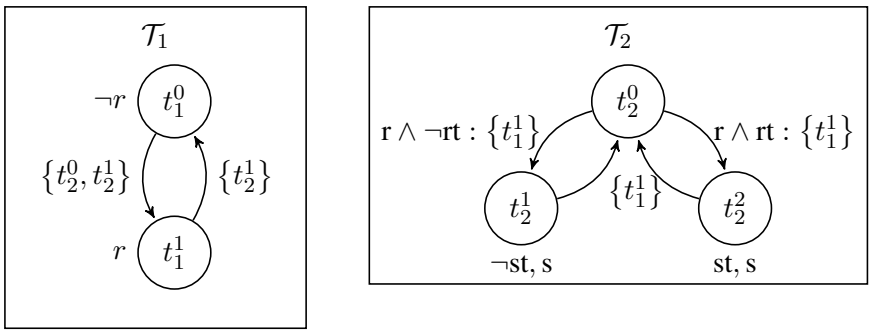


Figure 8.16: Disjunctive Example 2: Process template

The implementation shown in Figure 8.16 consists of one process template with 2 states, and one process template with 3 states. All output signals are lowered in the initial states t_1^0 and t_2^0. The implementation for template $\mathcal{T}_1$ is like in Example 1. The process can always move into the read state t_1^1, but can only move back to the initial state if there is at least one $\mathcal{T}_2$ process in one of the non-initial states. Processes belonging to template $\mathcal{T}_2$ can only move when process $\mathcal{T}_1^1$ is in state t_1^1. If allowed by the guard, instance of the second template grant requests depending on the request type rt, that is, they either move into state t_2^1 or state t_2^2.

Table 8.5 shows the runtimes. Because the number of (template-wise) assignments to label variables is smaller than the number of states in case of the second template, we expect that the label-based synthesis performs better than the state-based synthesis. For $n = 2$, we observe a small improvement. This can be explained by the fact that the cardinality of the guard variable set for the label-based encoding is only smaller by 1 compared to the state-based encoding. In this example, the cutoffs ($(1, 6)$ for single-indexed and multi-template double-indexed properties, $(1, 5)$ for deadlock detection) do not come into effect because the synthesis causes a timeout for $n \geq 3$. In case of label-based synthesis, one run out of 10 terminated within the timeout period (one run terminated in $2318s$, marked with (*) in the table), whereas all runs of the state-based approach yielded a timeout.

8.2.3 Conclusions

In case of conjunctive guards and small specifications, the cutoffs allow us to synthesize systems with an arbitrary size. Synthesis for conjunctive guarded systems benefits from the size-independent property cutoffs as well as the deadlock detection cutoffs which are less or equal to the bound for templates with ≤ 2 states. This enables a low runtime for synthesis in case of Example 3, which has a specification with 8 properties. Even for very small systems, synthesis for our disjunctive guarded examples does not terminate before the timeout period elapses. Comparing the timeout threshold for conjunctive and disjunctive guards shows that a timeout is given for much smaller systems (w.r.t. to the number of instances) in case of disjunctive guards. This is caused by having two process templates for disjunctive guard examples (instead of a single one as in the case of conjunctive guards), with an overall number of 4 states (Example 1), and 5 states (Example 2), respectively. In general, the primary cause for the steep increase in runtime is the size of the encoded UCT automata. For example, consider some specification (p1 – p2) and architecture properties (p3 – p4) of a small conjunctive guarded system.

(p1) $(\forall i : \mathsf{GF}\,\text{moves}^i) \rightarrow (\forall i : \mathsf{G}(r^i \rightarrow \mathsf{F}g^i))$

(p2) $\forall i, j : \mathsf{G}\,\neg(g^i \wedge g^j)$

(p3) $(\forall i : \mathsf{GF}\,\text{moves}^i) \rightarrow (\forall i : \mathsf{GF}\,\text{init}^i)$

(p4) $(\forall i : \mathsf{GF}\,\text{enabled}^i \rightarrow \mathsf{GF}\,\text{moves}^i) \rightarrow (\forall i : \mathsf{GF}\,\text{enabled}^i)$

Instances	(p1)	(p2)	(p3)	(p4)
2	5	2	8	11
3	6	2	14	36
4	7	2	22	109
5	8	2	32	318

Table 8.6: Impact of the system size on the size of UCT automata (without consideration of cutoffs)

Table 8.6 illustrates the correlation between system size (i.e., number of process instances) and UCT size for each property. We see that the symmetry considerations for specification properties (p1) and (p2) yield constant a number of UCT states (p2), and a small increase with respect to the system size (p2), respectively. These symmetry considerations do not apply to architecture properties. Hence, instantiating an architectural property for a higher number of processes has a significant impact on the size of the corresponding UCT automaton. Naturally, the steepness of the increase strongly depends on the structure of the translated property. Obviously, the deadlock detection property does not scale with respect to the system size, making synthesis for larger systems infeasible.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis we studied the class of disjunctive and conjunctive guarded systems described by Emerson and Kahlon [35]. We examined the already existing cutoffs for parameterized model checking of such systems [35], and investigated solutions for enabling efficient parameterized synthesis. Lifting the cutoff results for parameterized model checking to the synthesis domain as described in [48] is not possible easily because a) Emerson and Kahlon only consider systems without environment inputs (closed systems), and b) the existing cutoffs do not include any kind of fairness. We provided two solutions for parameterized synthesis. The first solution is to apply parameterized synthesis to closed systems, and then convert the synthesized closed system into an open one. This approach allows us to directly use the cutoff results identified by Emerson and Kahlon for model checking, however, suffers from state explosion, and lacks fairness. Because the used cutoffs depend on the number of states, state explosion results in exponentially increasing cutoffs, yielding a high synthesis runtime. Furthermore, if the aspect of fairness is ignored, this approach does not allow us to synthesize liveness properties. Therefore, we identified appropriate fairness constraints for the considered class, and then revisited the proofs of the cutoff results from [35]. Considering open systems and fairness, we proved new cutoffs for parameterized synthesis. Our results allow us to directly synthesize safety and liveness properties for the desired system.

We described logical formulae that represent the desired system implementation as a constraint system, allowing us to apply SMT-supported synthesis based on the semi-decision bounded synthesis approach by Finkbeiner and Schewe [38]. We implemented this algorithm in a Python prototype application (based on the parameterized synthesis tool PARTY [51]), and evaluated the runtime with respect to our cutoff results.

9.2 Future Work

The empirical evaluation shows that the new cutoffs enable parameterized synthesis of systems with small process templates. Nevertheless, the practical use is currently limited due to the high synthesis runtime for larger process templates as well as more complex specifications. We identified the size of the encoded UCT automata as a major contributor to the large number of constraints in the SMT instance, which has a significant impact on the synthesis runtime. One possibility to tackle this problem is to decrease the size of the formula that is converted into a UCT, for example by replacing the current fairness constraint by a more compact one. A further drawback is that the used cutoffs depend on the number of process template states. Finding implementations with a higher number of states is thus only possible under consideration of distributed systems with an increased number of processes. Obviously

this connection has a negative impact on the runtime. Hence, revisiting the cutoff proofs and finding cutoffs without this dependency or even constant cutoffs will be an important part of future research in this field.

Guarded protocols are not capable of modelling interesting real-world use cases, e.g., cache coherence protocols. However, they serve as a foundation for more advanced classes that allow to model such ubiquitous distributed systems. It is up to future work to establish efficient synthesis for such systems.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. “Recognizing Safety and Liveness”. In: *Distributed Computing* 2.3 (1987), pages 117–126 (cited on pages 18, 47).

[2] Benjamin Aminof et al. “Parameterized Model Checking of Rendezvous Systems”. In: *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. Edited by Paolo Baldan and Daniele Gorla. Volume 8704. Lecture Notes in Computer Science. Springer, 2014, pages 109–124. doi:10.1007/978-3-662-44584-6_9. http://dx.doi.org/10.1007/978-3-662-44584-6_9 (cited on page 8).

[3] Benjamin Aminof et al. “Parameterized Model Checking of Token-Passing Systems”. In: *VMCAI*. Edited by Kenneth L. McMillan and Xavier Rival. Volume 8318. Lecture Notes in Computer Science. Springer, 2014, pages 262–281 (cited on page 8).

[4] Keith Andrews. *Writing a Thesis: Guidelines for Writing a Master’s Thesis in Computer Science*. Graz University of Technology, Austria. Dec 2011. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page ix).

[5] Krzysztof R. Apt and Dexter Kozen. “Limits for Automatic Verification of Finite-State Concurrent Systems”. In: *Inf. Process. Lett.* 22.6 (1986), pages 307–309 (cited on page 7).

[6] Tamarah Arons et al. “Parameterized Verification with Automatically Computed Inductive Assertions”. In: *CAV*. Edited by Gérard Berry, Hubert Comon and Alain Finkel. Volume 2102. Lecture Notes in Computer Science. Springer, 2001, pages 221–234 (cited on page 7).

[7] Simon Außerlechner, Swen Jacobs and Ayrat Khalimov. *Tight Cutoffs for Guarded Protocols with Fairness*. 2015. arXiv: 1505.03273 (cited on pages 19, 21–24, 28, 29, 35, 38, 40–43).

[8] Tomás Babiak et al. “LTL to Büchi Automata Translation: Fast and More Deterministic”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Edited by Cormac Flanagan and Barbara König. Volume 7214. Lecture Notes in Computer Science. Springer, 2012, pages 95–109. doi:10.1007/978-3-642-28756-5_8. http://dx.doi.org/10.1007/978-3-642-28756-5_8 (cited on page 59).

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008 (cited on pages 12, 36, 37).

[10] Sergey Berezin, Sérgio Vale Aguiar Campos and Edmund M. Clarke. “Compositional Reasoning in Model Checking”. In: *Compositionality: The Significant Difference, International Symposium, COMPOS’97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*. Edited by Willem P. de Roever, Hans Langmaack and Amir Pnueli. Volume 1536. Lecture Notes in Computer Science. Springer, 1997, pages 81–102. doi:10.1007/3-540-49213-5_4. http://dx.doi.org/10.1007/3-540-49213-5_4 (cited on page 3).

[11] Gérard Berry, Hubert Comon and Alain Finkel, editors. *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Volume 2102. Lecture Notes in Computer Science. Springer, 2001.

[12] Armin Biere et al. "Bounded model checking". In: *Advances in Computers* 58 (2003), pages 117–148. doi:10.1016/S0065-2458(03)58003-2. http://dx.doi.org/10.1016/S0065-2458(03)58003-2 (cited on page 2).

[13] J. Richard Büchi and Lawrence H. Landweber. "Definability in the Monadic Second-Order Theory of Successor". In: *J. Symb. Log.* 34.2 (1969), pages 166–170 (cited on page 3).

[14] Jerry R. Burch et al. "Symbolic Model Checking: 10ˆ20 States and Beyond". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pages 428–439. doi:10.1109/LICS.1990.113767. http://dx.doi.org/10.1109/LICS.1990.113767 (cited on page 2).

[15] Rod M. Burstall. "Program Proving as Hand Simulation with a Little Induction". In: *IFIP Congress*. 1974, pages 308–312 (cited on page 2).

[16] Krishnendu Chatterjee et al. "Distributed synthesis for LTL fragments". In: *FMCAD*. IEEE, 2013, pages 18–25 (cited on page 3).

[17] Alonzo Church. "Logics, arithmetics, and automata". In: *Proceedings of the international congress of mathematicians, 1962*. Institut Mittag-Leffler, 1963, pages 23–35 (cited on page 3).

[18] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking*. Edited by Orna Grumberg and Helmut Veith. Volume 5000. Lecture Notes in Computer Science. Springer, 2008, pages 1–26 (cited on page 2).

[19] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. Edited by Dexter Kozen. Volume 131. Lecture Notes in Computer Science. Springer, 1981, pages 52–71. doi:10.1007/BFb0025774. http://dx.doi.org/10.1007/BFb0025774 (cited on pages 4, 8, 13).

[20] Edmund M. Clarke, E. Allen Emerson and A. Prasad Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *POPL*. Edited by John R. Wright et al. ACM Press, 1983, pages 117–126 (cited on page 2).

[21] Edmund M. Clarke, E. Allen Emerson and A. Prasad Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pages 244–263 (cited on page 2).

[22] Edmund M. Clarke, Orna Grumberg and David E. Long. "Model Checking and Abstraction". In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pages 1512–1542. doi:10.1145/186025.186051. http://doi.acm.org/10.1145/186025.186051 (cited on page 3).

[23] Edmund M. Clarke and Robert P. Kurshan, editors. *Computer Aided Verification, 2nd International Workshop, CAV '90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*. Volume 531. Lecture Notes in Computer Science. Springer, 1991.

[24] Edmund M. Clarke et al. "Verification by Network Decomposition". In: *CONCUR*. Edited by Philippa Gardner and Nobuko Yoshida. Volume 3170. Lecture Notes in Computer Science. Springer, 2004, pages 276–291 (cited on pages 7–9).

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Edited by C. R. Ramakrishnan and Jakob Rehof. Volume 4963. Lecture Notes in Computer Science. Springer, 2008, pages 337–340. doi:10.1007/978-3-540-78800-3_24. http://dx.doi.org/10.1007/978-3-540-78800-3_24 (cited on page 60).

[26] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Satisfiability Modulo Theories: An Appetizer". In: *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*. Edited by Marcel Vinicius Medeiros Oliveira and Jim Woodcock. Volume 5902. Lecture Notes in Computer Science. Springer, 2009, pages 23–36. doi:10.1007/978-3-642-10452-7_3. http://dx.doi.org/10.1007/978-3-642-10452-7_3 (cited on pages 14, 15).

[27] Leonardo Mendonça de Moura, Bruno Dutertre and Natarajan Shankar. "A Tutorial on Satisfiability Modulo Theories". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Edited by Werner Damm and Holger Hermanns. Volume 4590. Lecture Notes in Computer Science. Springer, 2007, pages 20–36. doi:10.1007/978-3-540-73368-3_5. http://dx.doi.org/10.1007/978-3-540-73368-3_5 (cited on page 14).

[28] Giorgio Delzanno. "Automatic Verification of Parameterized Cache Coherence Protocols". In: *CAV*. Edited by E. Allen Emerson and A. Prasad Sistla. Volume 1855. Lecture Notes in Computer Science. Springer, 2000, pages 53–68 (cited on page 9).

[29] E. Allen Emerson and Joseph Y. Halpern. "'Sometimes" and ''Not Never" revisited: on branching versus linear time temporal logic". In: *J. ACM* 33.1 (1986), pages 151–178. doi:10.1145/4904.4999. http://doi.acm.org/10.1145/4904.4999 (cited on page 12).

[30] E. Allen Emerson and Vineet Kahlon. "Model Checking Guarded Protocols". In: *LICS*. IEEE Computer Society, 2003, pages 361–370 (cited on pages 8, 9, 23).

[31] E. Allen Emerson and Vineet Kahlon. "Rapid Parameterized Model Checking of Snoopy Cache Coherence Protocols". In: *TACAS*. Edited by Hubert Garavel and John Hatcliff. Volume 2619. Lecture Notes in Computer Science. Springer, 2003, pages 144–159 (cited on page 8).

[32] E. Allen Emerson and Vineet Kahlon. "Parameterized Model Checking of Ring-Based Message Passing Systems". In: *CSL*. Edited by Jerzy Marcinkowski and Andrzej Tarlecki. Volume 3210. Lecture Notes in Computer Science. Springer, 2004, pages 325–339 (cited on page 7).

[33] E. Allen Emerson and Kedar S. Namjoshi. "Reasoning about Rings". In: *POPL*. Edited by Ron K. Cytron and Peter Lee. ACM Press, 1995, pages 85–94 (cited on page 9).

[34] E. Allen Emerson and Kedar S. Namjoshi. "On Reasoning About Rings". In: *Int. J. Found. Comput. Sci.* 14.4 (2003), pages 527–550 (cited on page 7).

[35] E.Allen Emerson and Vineet Kahlon. "Reducing Model Checking of the Many to the Few". In: *Automated Deduction - CADE-17*. Edited by David McAllester. Volume 1831. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pages 236–254. doi:10.1007/10721959_19. http://dx.doi.org/10.1007/10721959_19 (cited on pages 5, 8, 9, 22, 24–31, 35, 42, 43, 50, 54, 73).

[36] Javier Esparza, Alain Finkel and Richard Mayr. "On the Verification of Broadcast Protocols". In: *LICS*. IEEE Computer Society, 1999, pages 352–359 (cited on page 9).

[37] Bernd Finkbeiner and Sven Schewe. "Uniform Distributed Synthesis". In: *LICS*. IEEE Computer Society, 2005, pages 321–330 (cited on pages 3, 9, 19).

[38] Bernd Finkbeiner and Sven Schewe. “Bounded synthesis”. In: *STTT* 15.5-6 (2013), pages 519–539 (cited on pages 15, 16, 19, 20, 57, 73).

[39] Cormac Flanagan and Barbara König, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Volume 7214. Lecture Notes in Computer Science. Springer, 2012.

[40] Paul Gastin and Denis Oddoux. “Fast LTL to Büchi Automata Translation”. In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Edited by Gérard Berry, Hubert Comon and Alain Finkel. Volume 2102. Lecture Notes in Computer Science. Springer, 2001, pages 53–65. doi:10.1007/3-540-44585-4_6. http://dx.doi.org/10.1007/3-540-44585-4_6 (cited on pages 59, 61).

[41] Steven M. German and A. Prasad Sistla. “Reasoning about Systems with Many Processes”. In: *J. ACM* 39.3 (1992), pages 675–735. doi:10.1145/146637.146681. http://doi.acm.org/10.1145/146637.146681 (cited on page 9).

[42] Patrice Godefroid. “Using Partial Orders to Improve Automatic Verification Methods”. In: *Computer Aided Verification, 2nd International Workshop, CAV ’90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*. Edited by Edmund M. Clarke and Robert P. Kurshan. Volume 531. Lecture Notes in Computer Science. Springer, 1990, pages 176–185. doi:10.1007/BFb0023731. http://dx.doi.org/10.1007/BFb0023731 (cited on page 2).

[43] Alberto Griggio. “Introduction to SMT”. SAT/SMT summer school 2014. 2014 (cited on pages 14, 15).

[44] Sumit Gulwani et al. “Synthesis of loop-free programs”. In: *PLDI*. Edited by Mary W. Hall and David A. Padua. ACM, 2011, pages 62–73 (cited on page 9).

[45] Nicolas Halbwachs. “Synchronous Programming of Reactive Systems”. In: *CAV*. Edited by Alan J. Hu and Moshe Y. Vardi. Volume 1427. Lecture Notes in Computer Science. Springer, 1998, pages 1–16 (cited on page 2).

[46] D. Harel and A. Pnueli. “Logics and Models of Concurrent Systems”. In: edited by Krzysztof R. Apt. New York, NY, USA: Springer-Verlag New York, Inc., 1985. Chapter On the Development of Reactive Systems, pages 477–498. http://dl.acm.org/citation.cfm?id=101969.101990 (cited on page 1).

[47] C. Norris Ip and David L. Dill. “Better Verification Through Symmetry”. In: *Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL ’93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993*. Edited by David Agnew, Luc J. M. Claesen and Raul Camposano. Volume A-32. IFIP Transactions. North-Holland, 1993, pages 97–111 (cited on page 3).

[48] Swen Jacobs and Roderick Bloem. “Parameterized Synthesis”. In: *TACAS*. Edited by Cormac Flanagan and Barbara König. Volume 7214. Lecture Notes in Computer Science. Springer, 2012, pages 362–376 (cited on pages 9, 73).

[49] Swen Jacobs and Roderick Bloem. “Parameterized Synthesis”. In: *CoRR* abs/1401.3588 (2014) (cited on pages 9, 11, 16, 17, 19, 20).

[50] Maximilian Junker. “Using SMT Solvers for False-Positive Elimination in Static Program Analysis”. Master’s thesis. University of Augsburg, Ludwig Maximilian University of Munich, Technical University of Munich, 2010 (cited on page 15).

[51] Ayrat Khalimov, Swen Jacobs and Roderick Bloem. “PARTY Parameterized Synthesis of Token Rings”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Edited by Natasha Sharygina and Helmut Veith. Volume 8044. Lecture Notes in Computer Science. Springer, 2013, pages 928–933. doi:10.1007/978-3-642-39799-8_66. http://dx.doi.org/10.1007/978-3-642-39799-8_66 (cited on pages ix, 9, 47, 59, 73).

[52] Ayrat Khalimov, Swen Jacobs and Roderick Bloem. “Towards Efficient Parameterized Synthesis”. In: *VMCAI*. Edited by Roberto Giacobazzi, Josh Berdine and Isabella Mastroeni. Volume 7737. Lecture Notes in Computer Science. Springer, 2013, pages 108–127 (cited on pages 9, 38, 51, 57).

[53] Uri Klein, Nir Piterman and Amir Pnueli. “Effective Synthesis of Asynchronous Systems from GR(1) Specifications”. In: *VMCAI*. Edited by Viktor Kuncak and Andrey Rybalchenko. Volume 7148. Lecture Notes in Computer Science. Springer, 2012, pages 283–298 (cited on page 3).

[54] Fred Kröger. “LAR: A Logic of Algorithmic Reasoning”. In: *Acta Inf.* 8 (1977), pages 243–266. doi:10.1007/BF00264469. http://dx.doi.org/10.1007/BF00264469 (cited on page 2).

[55] Orna Kupferman, Nir Piterman and Moshe Y. Vardi. “Safraless Compositional Synthesis”. In: *CAV*. Edited by Thomas Ball and Robert B. Jones. Volume 4144. Lecture Notes in Computer Science. Springer, 2006, pages 31–44 (cited on page 3).

[56] Orna Kupferman and Moshe Y. Vardi. “Module Checking Revisited”. In: *CAV*. Edited by Orna Grumberg. Volume 1254. Lecture Notes in Computer Science. Springer, 1997, pages 36–47 (cited on page 7).

[57] Orna Kupferman and Moshe Y. Vardi. “Safraless Decision Procedures”. In: *FOCS*. IEEE Computer Society, 2005, pages 531–542 (cited on pages 3, 19).

[58] Orna Kupferman, Moshe Y. Vardi and Pierre Wolper. “Module Checking”. In: *Inf. Comput.* 164.2 (2001), pages 322–344 (cited on page 7).

[59] Leslie Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Trans. Software Eng.* 3.2 (1977), pages 125–143 (cited on page 18).

[60] Leslie Lamport. “Verification and Specifications of Concurrent Programs”. In: *REX School/Symposium*. Edited by J. W. de Bakker, Willem P. de Roever and Grzegorz Rozenberg. Volume 803. Lecture Notes in Computer Science. Springer, 1993, pages 347–374 (cited on page 1).

[61] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993 (cited on page 2).

[62] Greg Nelson and Derek C. Oppen. “Simplification by Cooperating Decision Procedures”. In: *ACM Trans. Program. Lang. Syst.* 1.2 (1979), pages 245–257. doi:10.1145/357073.357079. http://doi.acm.org/10.1145/357073.357079 (cited on page 15).

[63] Nir Piterman, Amir Pnueli and Yaniv Sa’ar. “Synthesis of Reactive(1) Designs”. In: *VMCAI*. Edited by E. Allen Emerson and Kedar S. Namjoshi. Volume 3855. Lecture Notes in Computer Science. Springer, 2006, pages 364–380 (cited on page 3).

[64] Amir Pnueli. “The Temporal Logic of Programs”. In: *FOCS*. IEEE Computer Society, 1977, pages 46–57 (cited on pages 2, 3, 13).

[65] Amir Pnueli and Roni Rosner. “On the Synthesis of a Reactive Module”. In: *POPL*. ACM Press, 1989, pages 179–190 (cited on page 3).

[66] Amir Pnueli and Roni Rosner. “Distributed Reactive Systems Are Hard to Synthesize”. In: *FOCS*. IEEE Computer Society, 1990, pages 746–757 (cited on pages 9, 19).

[67] Michael O. Rabin. “Decidability of second-order theories and automata on infinite trees.” In: *Trans. Amer. Math. Soc.* 141 (1969), pages 1–35 (cited on page 3).

[68] Sven Schewe and Bernd Finkbeiner. “Synthesis of Asynchronous Systems”. In: *LOPSTR*. Edited by Germán Puebla. Volume 4407. Lecture Notes in Computer Science. Springer, 2006, pages 127–142 (cited on page 19).

[69] Sven Schewe and Bernd Finkbeiner. “Bounded Synthesis”. In: *ATVA*. Edited by Kedar S. Namjoshi et al. Volume 4762. Lecture Notes in Computer Science. Springer, 2007, pages 474–488 (cited on pages 3, 5, 19, 45, 51).

[70] A. Prasad Sistla and Edmund M. Clarke. “The Complexity of Propositional Linear Temporal Logics”. In: *J. ACM* 32.3 (1985), pages 733–749 (cited on page 2).

[71] Luca Spalazzi and Francesco Spegni. “Parameterized Model-Checking for Timed-Systems with Conjunctive Guards (Extended Version)”. In: *CoRR* abs/1407.7305 (2014) (cited on page 8).

[72] Ichiro Suzuki. “Proving Properties of a Ring of Finite-State Machines”. In: *Inf. Process. Lett.* 28.4 (1988), pages 213–214. doi:10.1016/0020-0190(88)90211-6. http://dx.doi.org/10.1016/0020-0190(88)90211-6 (cited on pages 4, 7).

[73] Abhishek Udupa et al. “TRANSIT: specifying protocols with concolic snippets”. In: *PLDI*. Edited by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pages 287–296 (cited on page 9).

[74] Antti Valmari. “A Stubborn Attack On State Explosion”. In: *Computer Aided Verification, 2nd International Workshop, CAV ’90, New Brunswick, NJ, USA, June 18-21, 1990, Proceedings*. Edited by Edmund M. Clarke and Robert P. Kurshan. Volume 531. Lecture Notes in Computer Science. Springer, 1990, pages 156–165. doi:10.1007/BFb0023729. http://dx.doi.org/10.1007/BFb0023729 (cited on page 2).

[75] W.E. Wong et al. “Recent Catastrophic Accidents: Investigating How Software was Responsible”. In: *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*. Jun 2010, pages 14–22. doi:10.1109/SSIRI.2010.38 (cited on page 1).

[76] Jim Woodcock et al. “Formal methods: Practice and experience”. In: *ACM Comput. Surv.* 41.4 (2009). doi:10.1145/1592434.1592436. http://doi.acm.org/10.1145/1592434.1592436 (cited on page 1).