Dissertation

Reactive Synthesis: branching logic & parameter*i*zed systems

Ayrat Khalimov

Advisor:	Roderick Bloem Graz University of Technology, Austria
Reviewer:	Sven Schewe University of Liverpool, UK
Dean of Studies:	Denis Helic Graz University of Technology, Austria

Institute for Applied Information Processing and Communications
Graz University of Technology
A-8010 Graz, Austria
Januar 2018

Acknowledgements

This work would not be possible without RiSE network (established by Roderick Bloem and Helmut Veith). I stumbled upon the poster with the PhD position by chance, during a relaxed walk at EPFL where I was doing an internship.
I am grateful to my advisor Roderick Bloem, who honestly answered my questions, patiently directed me by asking questions and pitching ideas, and who always listened. Sasha Rubin showed how to be rigid and develop theories, with Swen Jacobs we wondered a lot around parameterised synthesis bouncing the token from each other, and Sven Schewe convinced me that tree automata are easy-peasy.
My colleagues Robert Könighofer, Georg Hofferek, and Bettina Könighofer helped in the initial integration and changed my attitude towards people outside of Russia. Our secretaries Martina Piewald, Melanie Blauensteiner, Ursula Urwanisch, and Angelika Wagner enabled me to focus on my work without administrative distractions.
Dedicated to my grandfather Rauf and our large family.

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 marked all material which has been quoted either literally or by content from the used sources.

.....	.....
place, date	signature

Abstract

Reactive synthesis is an automatic way to translate a human intention expressed in some logic into a system of some kind. This thesis has two parts, devoted to logic and to systems.

Part I. In 1963 Alonzo Church introduced the synthesis problem [28] for specifications in monadic second-order logic. Nowadays most model checkers and synthesizers use linear temporal logic (LTL) [70]. LTL reasons about system runs in a linear fashion. With LTL we can ask "does a run reach a particular state?" or "does a run visits a particular state infinitely often?". LTL is linear in its nature, leaving the designer without structural properties, which are expressible in computation tree logic (CTL) and its generalization CTL* [31, 41]. With CTL* we can ask "does a run never visit a particular state but it has a *possibility* to reach it?". Such properties are important—they allow for fine-tuning the system structure.

In Part I, we develop two new approaches to CTL* synthesis. The first approach is an extension (actually, two) of the SMT-based bounded synthesis [46]. We describe two extensions: one follows bottom-up CTL* model checking, another one follows the automata framework [63]. Then we develop the approach that reduces CTL* synthesis to LTL synthesis. The reduction turns any LTL synthesiser into a CTL* synthesiser. The approaches were implemented and are available online.

Part II. Modern systems become more and more distributed. Such distributed systems are typically parameterized by the number of processes: they should work for any number of processes. The parameterized synthesis problem [50] asks, given a parameterized specification, to find a process template, that can be cloned to form a correctly behaving system of any size. At the core of the method is the cutoff reduction technique: reduce reasoning about systems with an arbitrary number of processes to reasoning about systems of a fixed cutoff size. The intrinsic parameter, hidden in the parameterized synthesis problem, is how the processes are connected and how they communicate, i.e., the system architecture.

In Part II, we study parameterized synthesis for two system architectures. The first architecture is guarded systems [38] and is inspired by cache coherence protocols. In guarded systems, processes transitions are enabled or disabled depending on the existence of other processes in certain local states. The existing cutoff results [38] for guarded protocols are restricted to closed systems, and are of limited use for liveness properties. We close these gaps and prove tight cutoffs for open systems with liveness properties, and also cutoffs for detecting deadlocks.

The second architecture is token-ring systems [40], where the single token circulates processes arranged in a ring. The experiments with the existing parameterized synthesis method [50] showed that it does not scale to large specifications. First, we optimize the method by refining the cutoff reduction, using modularity and abstraction. The evaluation show several orders of magnitude speed-ups. Second, we perform parameterized synthesis case study on the industrial arbiter protocol AMBA [5]. We describe new tricks—a new cutoff extension and decompositional synthesis—that, together with the previously described optimizations, allowed us to synthesize AMBA in a parameterized setting, for the first time.

Contents

Chapter 1

Introduction

Dave: Hey, Elli, how can I calculate the week number from the date?
Elli: ...prints the C-function.
Dave: Great. Can this function also output three last requested dates?
Elli: ...prints another C-function.
Dave: Thanks. Can you also make it output the name of the requesting person?
Elli: I am sorry, Dave, I am afraid I can't do that.

In synthesis, we describe the required behaviour and ask the computer to find the solution with such a behaviour. (In the dialog above, Dave asks Elli to find a function that, given a date, outputs the week number to which the date belongs.) In *reactive* synthesis, we are interested not in simple "do-and-forget" functions, but rather in functions that interact with the user akin to functions with an internal state. (In the dialog above, the second C-function is reactive.) It is not always possible to find a solution, in which case the synthesizer (Elli) outputs "specification is unrealizable". (In the last dialogue request, the specification became unrealizable, because the person name is not available to the function to be synthesized.)

In 1963, Alonzo Church introduced the reactive synthesis problem [28]: given a formula in Monadic Second Order Logic of One Successor, and the inputs and the outputs of a circuit, find such a circuit such that all behaviors of the circuit satisfy the formula. (The circuit behaviour is an infinite string of inputs combined with the outputs.) Church's problem was solved by Rabin [75] and by Büchi and Landweber [25] in 1969.

Recent research in reactive synthesis focused on specifications given in Linear Temporal Logic (LTL), introduced by Pnueli [70] in 1977. LTL has temporal operators, like G (always) and F (eventually), and allows one to state properties like "every request is eventually granted": $\mathsf{G}(r \rightarrow \mathsf{F}\, g)$. A system satisfies a given LTL property if all its computations satisfy it. Pnueli and Rosner proved [71] that the LTL synthesis problem is 2EXPTIME-complete. Their approach translates a given LTL formula into a nondeterministic Büchi automaton, then determinises it into a deterministic parity automaton with the aid of involved Safra construction [77], turns the automaton into a game, and solves the game. Recent research focused on how to overcome the high complexity and Safra construction: the work [15] considered the synthesis for a subset of LTL called GR(1), the work [46, 61] considered

bounding the system size and gave a name to Bounded Synthesis, by combining the previous bounding with efficient data structures—Anti-chains Synthesis [45]. The SYNTCOMP competition [1] is another recent initiative with the goal to advance efficient synthesisers and popularise reactive synthesis.

Despite substantial progress, reactive synthesis is not as widespread as model checking. The major reason, I believe, is that writing the specifications—especially *complete* specifications—is hard. The issue is less pronounced in model checking, because we do not need all the properties, only those to model check.

In light of this issue, there are two directions to proceed. First, we can develop synthesis approaches for richer logics, which can ease writing the specifications. Second, we can find application contexts where high specification costs are acceptable. This thesis targets both directions: we develop new synthesis approaches for the logic called CTL*, and we delve into synthesis of distributed algorithms.

Part I: Excursion into Branching Logic

Computation Tree Logic (CTL) [31] was introduced by Emerson and Clarke in 1981 to circumvent the high complexity (PSPACE-complete) of the LTL model checking problem and to be able to specify structural properties. In 1986 Emerson and Halpern introduced a generalization, Computation Tree Star Logic (CTL*) [41], that subsumes both CTL and LTL.

In contrast to LTL, which reasons about (linear) computation runs, CTL* reasons about (branching) computation trees. We can get such a tree by unfolding the system transition structure. CTL* has—in addition to temporal operators—path quantifiers: A (on all paths) and E (there exists a path). Such path quantifiers allow us to reason about branching structure of trees, not just about their "linear" paths. For example, CTL* formula "$\mathsf{AGEF}\,reset$" says: "on all tree paths, from every tree node, there should be a path into a node where 'reset' holds". We cannot express such a property using LTL alone.

Despite CTL* being more expressible than LTL, the complexity of CTL* synthesis (2EXPTIME-complete) stays the same. This prompted us to look into approaches to CTL* synthesis.

The standard solution [62] to CTL* synthesis turns the CTL* formula into an alternating hesitant tree automaton, removes nondeterminism and derives a universal co-Büchi tree automaton, determinises it using Safra construction [77] into a parity tree automaton, and, finally, checks its non-emptiness. If it is empty, then the specification is unrealisable, otherwise we can extract the system from the proof of the non-emptiness. This approach is hard to implement correctly and efficiently, due to the involved Safra construction[1].

Part I contribution is two practical approaches to CTL* synthesis.

[1]It was a common belief that the Safra construction is difficult to implement and results in impractical algorithms. However, the belief might be wrong, as SYNTCOMP [1] in 2017 showed: the LTL synthesiser `ltl-synt` that used Safra construction performed very well.

Contribution I.1: CTL* Bounded Synthesis

We developed two bounded synthesis approaches for the CTL* specifications. Let us recall how the SMT-based bounded synthesis by Schewe and Finkbeiner [46] works: we bound the system size, and encode the resulting synthesis problem into an SMT query[2]. The query encodes the model checking question: whether a system—which is yet unknown—is accepted by the automaton. Bounding the system size makes it possible to encode such a model checking query into an SMT query. To solve such a query, an SMT solver efficiently enumerates every possible system of a given size, and checks if it is correct. Thus, if the SMT query is satisfiable, then we extract the system (of the given size), otherwise increase the system size and repeat. The loop stops when the bound on the system size—provided by the user or from the theory—is reached.

Our first bounded synthesiser for CTL* resembles bottom-up CTL* model checking [9]: it introduces an atom for each subformula of the CTL* formula, and encodes into an SMT query whether the atom holds in a system state, for every state. We also require the top-level atom, representing the whole CTL* formula, to hold in the initial system state, Hence, if the SMT query is satisfiable, then there is a system of the given size, which satisfies the CTL* formula. Otherwise, increase the system size and repeat.

Our second bounded synthesiser for CTL* uses the automata framework [63]: translate the CTL* formula into an alternating hesitant automaton, then encode into an SMT query whether there is a system of a given size that is accepted by the automaton. Conceptually, the approach is the same as the previous one, except that we do not introduce atoms for subformulas explicitly and instead use their automata representation.

The results constitute Chapter 3 and were published in:

[56] *Bounded Synthesis for Streett, Rabin, and CTL**, by Ayrat Khalimov and Roderick Bloem, at CAV conference, 2017

Contribution I.2: CTL*-via-LTL Synthesis

We reduce synthesis for CTL* properties to synthesis for LTL. In the context of model checking this is impossible—CTL* is more expressive than LTL. Yet, in synthesis we have knowledge of the system structure *and* we can add new outputs. These outputs can be used to encode witnesses of the satisfaction of CTL* subformulas directly into the system. This way, we construct an LTL formula, over old and new outputs and original inputs, which is realisable if, and only if, the original CTL* formula is realisable. The CTL*-via-LTL synthesis approach preserves the problem complexity, although it might produce systems that are larger than necessary. Furthermore, the approach directly benefits from the performance advances of LTL synthesisers. The results constitute Chapter 4 and were published in:

[2]Satisfiability Modulo Theory (SMT) [13] query is a set of constraints over in a given theory. For example, in Linear Integer Arithmetic theory, the constraints talk about integer variables, use operations plus, minus, and the comparison relations. Such a query asks whether there are values for integer variables that make the constraint true.

[23] *CTL* Synthesis via LTL Synthesis*, by Roderick Bloem and Sven Schewe and Ayrat Khalimov, at SYNT workshop, 2017

Part II: Excursion into Parameterized Systems

Modern systems become more and more distributed. Distributed systems are hard to implement and even harder to debug. Yet, the failure of such systems may be unacceptable. Thus, substantial efforts are devoted to ensure the correctness of distributed systems. In Part II, we look into the hard task of automatic synthesis of distributed parameterized systems.

Most distributed systems, algorithms, and data structures are *parameterized*: they should work for a varied, not a priori fixed, number of the components. The parameterized synthesis problem [50] asks, given a parameterized specification, to find a process template, that can be cloned to form a correctly behaving system of any size. An example parameterized specification is:

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \wedge \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i). \end{array}$$

The synthesizer should find a process template, having input r and output g, such that a system composed of any number of such processes, satisfies the above specification. The related question is that of parametrized model checking where the process template is given. The intrinsic parameter, hidden in the parameterized synthesis problem, is how the processes are connected and how they communicate, i.e., the system architecture. The survey of existing cutoff and decidability results for many different system architectures can be found in [21]. We focus on two system architectures: guarded systems and token-ring systems.

A common approach to solve the parameterized synthesis and model checking problems is to use the cutoff reduction [40]: reduce reasoning about systems with an arbitrary number of processes to reasoning about systems of a fixed cutoff size. For example, if we consider the parameterized specification mentioned above and token-ring systems, then it is enough to consider a system with 4 processes: if it is correct, then any larger system is correct.

Contribution II.1: Cutoffs for Parameterized Guarded Systems

Guarded systems [38] are inspired by cache coherence protocols found in most modern processors. A cache coherence protocol is usually described by states, where transitions between states happen depending on whether or not there is a processor in a particular state. I.e., the transitions are guarded. Inspired by this, in guarded systems, processes transitions are enabled or disabled depending on the existence of other processes in certain local states. Our contribution concerns both parameterized synthesis and parameterized verification. Our work stems from the observation that existing cutoff results for guarded systems (i) are restricted to closed systems, and (ii) are of limited use for liveness properties because reductions do not preserve fairness. We close these gaps and obtain new cutoff results for open systems with liveness properties under fairness assumptions. Furthermore,

we obtain cutoffs for the detecting deadlocks, which are of paramount importance in synthesis. Finally, we prove tightness or asymptotic tightness for the new cutoffs. The results constitute Chapter 5 and were published in:

[7] *Tight Cutoffs for Guarded Protocols with Fairness*, by Simon Außerlechner and Swen Jacobs and Ayrat Khalimov, at VMCAI conference, 2016

Contribution II.2: Case Study of Parameterized Token-ring AMBA

In token-ring systems, a single token circulates in the system. A process possessing the token knows that no other process has the token. Based on this information, the process can, for example, raise the grant signal. If all processes raise the grant only when they posses the token, then the grants will be mutually exclusive. Thus, the token serves as the resource token.

The experiments with the existing parameterized synthesis method [50] showed that it does not scale to large specifications. First, we optimize the method by refining the cutoff reduction. The experiments show speed-ups of several orders of magnitude. Second, we perform parameterized synthesis case study on the industrial arbiter protocol AMBA [5]. We describe new cutoff extension and decompositional synthesis tailored to AMBA that, together with the previously mentioned optimizations, allowed us to synthesize AMBA in parameterized setting, for the first time. The results constitute Chapter 6 and were published in:

[54] *Towards Efficient Parameterized Synthesis*, by Ayrat Khalimov and Swen Jacobs and Roderick Bloem, at VMCAI conference, 2013

[57] *PARTY: Parameterized Synthesis of Token Rings*, by Ayrat Khalimov and Swen Jacobs and Roderick Bloem, at CAV conference, 2013

[20] *Parameterized Synthesis Case Study: AMBA AHB*, by Ayrat Khalimov and Swen Jacobs and Roderick Bloem, at SYNT workshop, 2014

Other Results

Here are the results that did not make their way into the thesis:

[21] *Decidability of Parameterized Verification*, book of 170 pages, by Roderick Bloem and Swen Jacobs and Ayrat Khalimov and Igor Konnov and Sasha Rubin and Helmut Veith and Josef Widder.
In this book we consider the important case of systems parameterized by the number of processes in the system and where each process is independent of that number. The literature in this area produced a wealth of computational models for systems based on token passing, broadcast communication, guarded transitions, and other communication primitives. We introduce a computational model that unites the central synchronization and communication primitives of many models. We survey existing decidability and undecidability results, and provide a systematic overview of the basic problems in this research area.

[22] *Decidability in Parameterized Verification*, the journal version of the above book; appeared in SIGACT News in 2016.

[55] *Specification Format for Reactive Synthesis Problems*, by Ayrat Khalimov, at SYNT workshop, 2015.
To do synthesis, we need a specification. Writing specifications is hard. In this paper, we propose a user-friendly format to ease the specification work, in particularly, that of specifying partial implementations. Also, we provide scripts to convert specifications in the new format into the SYNTCOMP format, thus benefiting from state of the art synthesizers.

[3] *Parameterized Model Checking of Token-Passing Systems*, by Benjamin Aminof and Swen Jacobs and Ayrat Khalimov and Sasha Rubin, at VMCAI conference, 2014.
In this paper, we revisit the parameterized model checking problem for token-passing systems and specifications in indexed CTL*\X. We unify and substantially extend the results of Emerson and Namjoshi [39, 40] and Clarke et al. [30] by systematically exploring fragments of indexed CTL*\X with respect to general network topologies. For each fragment we establish whether a cutoff exists, and for some concrete topologies, such as rings, cliques and stars, we infer small cutoffs. Finally, we show that the problem becomes undecidable, and thus no cutoffs exist, if processes are allowed to choose the directions in which they send or from which they receive the token.

[58] *OpenSEA: Semi-Formal Methods for Soft Error Analysis*, by Patrick Klampfl and Robert Könighofer and Roderick Bloem and Ayrat Khalimov and Aiman Abu-Yonis and Shiri Moran, on arxiv, 2017.
Due to alpha-particles and cosmic rays, modern circuits are prone to bit flips. To alleviate the problem, designers develop protection circuits, but they are hard to implement right. This leads to bugs: an undetected fault can bring miscalculations, the protection that alarms about harmless faults incurs performance penalty. In this paper, we use formal methods on designers input tests, while keeping time-location open. This idea is at the core of the tool OpenSEA. OpenSEA can (i) find latches vulnerable to and protected against faults, (ii) find tests that exhibit checker false alarms, (iii) use fixed and open inputs, and (iv) use environment assumptions. Evaluation on a number of industrial designs shows that OpenSEA produces valuable results.

Part I

Excursion Into Branching Logic

Approaches to CTL* Synthesis

Overview of Part I

The reactive synthesis problem was introduced by Alonzo Church [28]. Given a specification as a formula in Monadic Second Order Logic of One Successor (MSO), the question is to produce a circuit such that *all* its behaviors satisfy the formula. Later Pnueli introduced Linear Temporal Logic (LTL) [70] and together with Rosner solved the synthesis problem for LTL [71]. Now LTL is the main basic logic for specifications. Both these logics, MSO and LTL, are *linear*: they describe the set of behaviours, but do not allow for specifying *structural* properties of the systems.

To be able to specify structural properties (and to circumvent a relatively high complexity of the verification wrt. LTL), Emerson and Clarke introduced Computation Tree Logic (CTL) [31]. Later Emerson and Halpern introduced Computation Tree Star Logic (CTL*) [41] that subsumed both CTL and LTL.

Let us briefly compare LTL and CTL*.

LTL reasons about *computations*. The logic has *temporal* operators, e.g., G (always) and F (eventually), and can describe properties like "every request is eventually granted": $\mathsf{G}(r \rightarrow \mathsf{F}\, g)$. A system satisfies such an LTL property iff *all* its computations satisfy it. Thus a system is characterized by its computations.

In contrast, CTL* reasons about computation *trees*. Thus, a system is viewed as a tree (cf. set of linear paths for LTL), and we can get such a tree by unfolding the system. CTL* has—in addition to temporal operators—*path quantifiers*: A (on all paths) and E (there exists a path). Such path quantifiers allow us to reason about branching structure of trees, not just about the set of its "linear" paths. For example, the CTL* formula "$\mathsf{AGEF}\, reset$" says: "on all tree paths, from every tree node, there should be a path into a node where 'reset' holds". We cannot express such a property using LTL alone.

This part of the thesis explores synthesis approaches from properties in CTL*. It consists of two chapters.

In Chapter 3 we introduce two approaches to synthesis from CTL*. Both approaches follow the Bounded Synthesis approach introduced by Finkbeiner and Schewe [46]. In Bounded Synthesis, we repeatedly search for a system of increasing sizes, until we find a solution. Bounded Synthesis is very flexible and can be easily adapted to do e.g. distributed synthesis. We extend Bounded Synthesis to specifications in CTL* and beyond.

The disadvantage of Bounded Synthesis is that it is susceptible to system size: it works well when the specification admits a small implementation, but less well when no small implementation exists. The same holds for our CTL* Bounded Synthesis.

In Chapter 4, partly to overcome this disadvantage, we introduce a reduction of the CTL* synthesis problem to the LTL synthesis problem. After applying the reduction, any *LTL* synthesiser can do CTL^* synthesis. Notice that for model checking such a reduction is impossible—CTL* is more expressive than LTL. Yet, in synthesis we control the system structure, which enables the reduction. The CTL*-via-LTL synthesis approach preserves the problem complexity, although it might increase the size of a system.

The approaches differ in how they ensure the satisfaction of existential CTL* subformulas (recall that universal CTL* subformulas, just like LTL, talk about system paths as a whole, while existential CTL* subformulas specify the existence of a system path). Recall from Section 2 that bounded synthesis encodes the LTL synthesis problem into the SMT satisfaction problem. The SMT constraints annotate the states of a *product* (of a yet unknown system with an automaton expressing a given CTL* formula) with information that ensures that all lassos in the product are not "bad" (for universal subformulas) and that there are "good" lassos (for existential subformulas). In contrast, CTL*-via-LTL synthesis produces an LTL formula that talks about *system* paths and has no direct access to the product. Hence we move annotations into a system which may increase its size.

This thesis part is organized as follows. In the next Chapter 2 we introduce the definitions which are used in both chapters. Chapter 3 focuses on extensions of Bounded Synthesis to CTL*, while Chapter 4 describes the CTL*-to-LTL synthesis reduction. Both chapters depend on the definitions section, but are independent of each other.

Chapter 2

Common Definitions for Part I

Notation: $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ is the set of Boolean values, $\mathbb{N}$ is the set of natural numbers (excluding 0), $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, $[k]$ is the set $\{i \in \mathbb{N} \mid i \leq k\}$ and $[0, k]$ is the set $[k] \cup \{0\}$ for $k \in \mathbb{N}$.

The powerset of A is denoted by 2^A. We often write (a, x) instead of $a \cup x$ (that is from $2^{A \cup X}$), and $a \cup x$ instead of (a, x) (that is from $2^A \times 2^X$), when $a \in 2^A$, $x \in 2^X$ and $A \cap X = \emptyset$.

We denote substitution by the symbol $\mapsto$. E.g., $(a \wedge b)[a \mapsto x]$ is $x \wedge b$.

All systems and automata are finite, paths are infinite, and trees have only infinitely long paths but are finitely-branching—unless explicitly stated.

2.1 Moore Systems

A *(Moore) system* M is a tuple $(I, O, T, t_0, \tau, out)$ where I and O are disjoint sets of input and output variables, T is the set of states, $t_0 \in T$ is the initial state, $\tau : T \times 2^I \to T$ is a transition function, $out : T \to 2^O$ is the output function that labels each state with a set of output variables. Note that systems have no dead ends and have a transition for every input. We write $t \xrightarrow{io} t'$ when $t' = \tau(t, i)$ and $out(t) = o$. We abuse the notation and define $\tau(t, w)$ for $w_1 w_2 ... w_n \in (2^I)^+$ to be the system state t_n such that $t_0 \xrightarrow{io_0} t_1 \xrightarrow{io_1} ... \xrightarrow{io_{n-1}} t_n$, i.e., $\tau(t, w)$ is the state where the system ends after reading the word w, when starting from the initial state.

A *system path* is a sequence $t_1 t_2 ... \in T^\omega$ such that for every $i \in \mathbb{N}$ there is $e \in 2^I$ with $\tau(t_i, e) = t_{i+1}$. An *input-labeled system path* is a sequence $(t_1, e_1)(t_2, e_2)... \in (T \times 2^I)^\omega$ where $\tau(t_i, e_i) = t_{i+1}$ for every $i \in \mathbb{N}$. We sometimes use notation $t_1 \xrightarrow{e_1} t_2 \xrightarrow{e_2} t_3 ...$ to describe the input-labeled system path $(t_1, e_1)(t_2, e_2)....$ A *system computation starting from* $t_1 \in T$ is a sequence $(o_1 \cup e_1)(o_2 \cup e_2)... \in (2^{I \cup O})^\omega$ for which there exists an input-labeled system path $(t_1, e_1)(t_2, e_2)...$ and $o_i = out(t_i)$ for every $i \in \mathbb{N}$. We write *system computation* to mean system computation starting from the initial state. Note that since systems are Moore, the output o_i cannot "react" to input e_i—the outputs are "delayed" with respect to inputs.

Remark 1. There are two ways to group inputs and outputs into computations. The first way is to introduce an initial transition $\tau_I : 2^I \to T$ instead of using the initial state t_0. Then the input-labeled system path $\xrightarrow{e_1} t_1 \xrightarrow{e_2} t_2 \xrightarrow{e_3} t_3 ...$ corresponds to the computation $(e_1, out(t_1))(e_2, out(t_2))(e_3, out(t_3))....$ Another way is to avoid

using the initial transition—use the initial state t_0 instead—and "shift" inputs and outputs. Then an input-labeled system path $t_0 \stackrel{e_1}{\to} t_1 \stackrel{e_2}{\to} t_2...$ corresponds to the computation $(out(t_0), e_1)(out(t_1), e_2)....$ We use the second approach.

2.2 Trees

A *(infinite) tree* is a tuple $(D, L, V \subseteq D^*, l : V \to L)$, where

- D is the set of directions (in our case, finite),
- L is the set of node labels (in our case, finite),
- V is the (infinite) set of nodes satisfying: (i) $\epsilon \in V$ is called the root (the empty sequence), (ii) V is closed under prefix operation (i.e., every node is connected to the root), (iii) for every $n \in V$ there exists a $d \in D$ such that $n \cdot d \in V$ (i.e., there are no leafs),
- l is the node labeling function.

A tree (D, L, V, l) is *exhaustive* iff $V = D^*$. A tree is *non-labeled* iff $|L| = 1$ and then we omit L and l.

A *tree path* is a sequence $n_1 n_2... \in V^\omega$, such that, for every i, there is $d \in D$ such that $n_{i+1} = n_i \cdot d$.

In contexts where I and O are inputs and outputs, we call an exhaustive tree (D, L, V, l) a *computation tree*, where $D = 2^I$, $L = 2^O$, $V = D^*$, and $l : V \to 2^O$. We omit D and L when they are clear from the context.

With every system $M = (I, O, T, t_0, \tau, out)$ we associate the computation tree (D, L, V, l) such that, for every $n \in V$: $l(n) = out(\tau(t_0, n))$. We call such a tree a *system computation tree*.

A computation tree is *regular* iff it is a system computation tree for some (finite) system.

Two Views on the System

Later we introduce logics CTL* and LTL to distinguish correct from buggy systems. The two logics look at systems from two sides.

On one side, we can associate with a system M a set of its computations $b(M) \subseteq (2^{I \cup O})^\omega$. A formula φ in Linear Temporal Logic (LTL) (introduced later) describes a set of infinite words $L(\varphi)$. Thus, we can use an LTL formula to specify all correct computations. Then a system M is correct wrt. LTL formula φ iff $b(M) \subseteq L(\varphi)$, i.e., all system computations satisfy φ.

On the other side, we might want to specify structural properties of systems. E.g., whether from every system state we can branch into a state satisfying p and we can branch into a state satisfying $\neg p$. In this case, characterizing a system by its set of computations—e.g. using LTL—is not possible. Instead, we associate with a system its computation tree. A formula Φ in Computation Tree Logic (defined later) describes a set of computation trees $L(\Phi)$. Thus, we can use such

a formula to describe a set of all correct computation trees. Then a system M is correct wrt. Φ iff $(V, l) \in L(\Phi)$, i.e., the system computation tree satisfies Φ.

2.3 Logics: CTL* with Inputs and LTL

CTL* with inputs (release PNF)

Fix two disjoint sets: inputs I and outputs O. Below we define CTL* with inputs, in release positive normal form[1]. The definition differentiates inputs and outputs (see Remark 2).

Syntax. *State formulas* have the grammar:

$$\Phi = \mathsf{true} \mid \mathsf{false} \mid o \mid \neg o \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathsf{A}\varphi \mid \mathsf{E}\varphi$$

where $o \in O$ and φ is a path formula. *Path formulas* are defined by the grammar:

$$\varphi = \Phi \mid i \mid \neg i \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \varphi \,\mathsf{R}\, \varphi,$$

where $i \in I$. The temporal operators G and F are defined as usual.

The above grammar describes the CTL* formulas in positive normal form. The general CTL* formula (in which negations can appear anywhere) can be converted into the formula of this form with no size blowup, using the equivalence $\neg(a \,\mathsf{U}\, b) \equiv \neg a \,\mathsf{R}\, \neg b$ and some others.

Semantics. We define the semantics of CTL* with respect to a computation tree (V, l) (where $D = 2^I$ and $L = 2^O$). The definition is very similar to the standard one [9], except for a few cases involving inputs (marked with "+").

Let $n \in V$ and $o \in O$. Then:

- $n \not\models \Phi$ iff $n \models \Phi$ does not hold,
- $n \models \mathsf{true}$ and $n \not\models \mathsf{false}$,
- $n \models o$ iff $o \in l(n)$, $n \models \neg o$ iff $o \notin l(n)$,
- $n \models \Phi_1 \wedge \Phi_2$ iff $n \models \Phi_1$ and $n \models \Phi_2$. Similarly for $\Phi_1 \vee \Phi_2$.
- $n \models \mathsf{A}\varphi$ iff for all tree paths π starting from n: $\pi \models \varphi$. For $\mathsf{E}\varphi$, replace "for all" with "there exists".

Let $\pi = n_1 n_2 ... \in V^\omega$ be a tree path, $i \in I$, and $n_2 = n_1 \cdot e$ where $e \in 2^I$. For $k \in \mathbb{N}$, define $\pi_{[k:]} = n_k n_{k+1} ...$, i.e., the suffix of π starting in n_k. Then:

- $\pi \models \Phi$ iff $n_1 \models \Phi$,

\+ $\pi \models i$ iff $i \in e$, $\pi \models \neg i$ iff $i \notin e$,

[1] This form is sometimes called negation normal form. For the name, we follow [9]. Note that without the release operator R—the dual of the until operator U—the logic is less expressive due to the restriction on negations. That explains the name "release PNF".

- $\pi \models \varphi_1 \wedge \varphi_2$ iff $\pi \models \varphi_1$ and $\pi \models \varphi_2$. Similarly for $\varphi_1 \vee \varphi_2$.
- $\pi \models \mathsf{X}\,\varphi$ iff $\pi_{[2:]} \models \varphi$,
- $\pi \models \varphi_1 \,\mathsf{U}\, \varphi_2$ iff $\exists l \in \mathbb{N} : (\pi_{[l:]} \models \varphi_2 \wedge \forall m \in [1, l-1] : \pi_{[m:]} \models \varphi_1)$,
- $\pi \models \varphi_1 \,\mathsf{R}\, \varphi_2$ iff $(\forall l \in \mathbb{N} : \pi_{[l:]} \models \varphi_2) \vee (\exists l \in \mathbb{N} : \pi_{[l:]} \models \varphi_1 \wedge \forall m \in [1, l] : \pi_{[m:]} \models \varphi_2)$.

A *computation tree* (V, l) *satisfies a CTL* state formula* Φ, written $(V, l) \models \Phi$, iff the root node satisfies it. A *system* M *satisfies a CTL* state formula* Φ, written $M \models \Phi$, iff its computation tree satisfies it.

Remark 2 (Subtleties)**.** Note that $(V, l) \models i \wedge o$ is not defined, since $i \wedge o$ is not a state formula. Let $r \in I$ and $g \in O$. By the semantics, $\mathsf{E}r \equiv \mathsf{true}$ and $\mathsf{E}\neg r \equiv \mathsf{true}$, while $\mathsf{E}g \equiv g$ and $\mathsf{E}\neg g \equiv \neg g$. These facts are the consequences of the way we group inputs with outputs (see also Remark 1).

LTL

The syntax of LTL formulas (in general form) is:

$$\phi = \mathsf{true} \mid p \mid \neg p \mid \phi \wedge \phi \mid \neg\phi \mid \phi \,\mathsf{U}\, \phi \mid \mathsf{X}\,\phi,$$

where $p \in I \cup O$. The temporal operators G and F are defined as usual, and $\mathsf{false} = \neg\mathsf{true}$. The semantics is standard (see, e.g., [9]). A computation tree (V, l) satisfies an LTL formula ϕ, written $(V, l) \models \phi$, iff all tree paths starting in the root satisfy it. A system satisfies an LTL formula iff its computation tree satisfies it (equivalently, every system computation starting from the initial state satisfies the LTL formula).

2.4 Tree Automata

Tree automata consume infinite trees and output "accept" or "reject". Since every Moore system has a corresponding computation tree, tree automata can be used to differentiate buggy Moore machines from correct ones. Also, CTL* can be translated into a special type of alternating tree automata. Thus, tree automata are the excellent tool for model checking and synthesis.

We start with a general definition of alternating tree automata, then introduce different acceptance conditions, then introduce alternating hesitant tree automata.

Notation $\mathcal{B}^+(S)$. For a finite non-empty set S, let $\mathcal{B}^+(S)$ be the set of all positive Boolean formulas over elements of S, i.e., every such a formula ϕ has the syntax: $\phi = e \mid \phi \wedge \phi \mid \phi \vee \phi$, where $e \in S$. Note that $\mathsf{false} \notin \mathcal{B}^+(S)$ and $\mathsf{true} \notin \mathcal{B}^+(S)$. As we will see later, these are not limitations in our context. Also, since the set S is finite, any Boolean formula over atoms in S and which is semantically different from true and false is equivalent to some formula in $\mathcal{B}^+(S)$. Furthermore, every formula $\phi \in \mathcal{B}^+(S)$ can be rewritten into formula $\phi' \in \mathcal{B}^+(S)$ in disjunctive normal form (DNF) or into formula $\phi'' \in \mathcal{B}^+(S)$ in conjunctive normal form (CNF). We assume that formulas in $\mathcal{B}^+(S)$ (and thus CNF and DNF formulas) have neither redundant atoms, conjuncts, nor disjuncts.

Alternating tree automata

An *alternating tree automaton* is a tuple $(\Sigma, D, Q, q_0, \delta, acc)$, where Σ is the set of node propositions, D is the set of directions, $q_0 \subseteq Q$ is the initial state, $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$ is the transition relation, and $acc : Q^\omega \to \mathbb{B}$ is an acceptance condition. For simplicity we assume that δ is total wrt. directions. Thus, it is worth noting about $\delta(q, \sigma)$, for every $(q, \sigma) \in Q \times \Sigma$:

- if we rewrite $\delta(q, \sigma)$ into DNF, then each conjunct mentions each direction at least once ("totalness" wrt. directions).
- $\delta(q, \sigma) \neq \mathsf{false}$ and $\delta(q, \sigma) \neq \mathsf{true}$. These are not limitations, because we can emulate true and false by introducing additional states and modifying *acc*.

The above means that δ has a transition for every possible argument and direction.

Fix two disjoint sets, inputs I and outputs O.

Tree automata consume exhaustive trees like $(D, L = \Sigma, V = D^*, l : V \to \Sigma)$ and produce run-trees.

A *run-tree* of an alternating tree automaton $(\Sigma = 2^O, D = 2^I, Q, q_0, \delta, acc)$ on a computation tree $(V = (2^I)^*, l : V \to 2^O)$ is a tree with directions $2^I \times Q$, labels $V \times Q$, nodes $V' \subseteq (2^I \times Q)^*$, labeling function l' such that

- $l'(\epsilon) = (\epsilon, q_0)$,
- if $v \in V'$ with $l'(v) = (n, q)$, then:
  there exists $\{(d_1, q_1), ..., (d_k, q_k)\}$ that satisfies $\delta(q, l(n))$ and $n \cdot (d_i, q_i) \in V'$ for every $i \in [1, k]$.

Intuitively, we run the alternating tree automaton on the computation tree:

(1) We mark the root node of the computation tree with the automaton initial state q_0. We say that initially, in the node ϵ, there is only one copy of the automaton and it has state q_0.

(2) We read the label $l(n)$ of the current node n of the computation tree and consult the transition function $\delta(q, l(n))$. The latter gives a set of conjuncts of atoms of the form $(d', q') \in D \times Q$. We nondeterministically choose one such conjunction $\{(d_1, q_1), ..., (d_k, q_k)\}$ and send a copy of the alternating automaton into each direction d_i in the state q_i. Note that we can send up to $|Q|$ copies of the automaton into one direction (but into different automaton states). That is why a run-tree defined above has directions $2^I \times Q$ rather than 2^I.

(3) We repeat step (2) for every copy of the automaton. As a result we get a run-tree: a tree labeled with nodes of the computation tree and states of the automaton.

A *run-tree is accepting* iff every run-tree path starting from the root is accepting. A run-tree path $v_1 v_2 ...$ is accepting iff $acc(q_1 q_2 ...)$ holds (*acc* is defined later), where q_i for every $i \in \mathbb{N}$ is the automaton state part of $l'(v_i)$. Note that every

run-tree path is infinite. (In particularly, we do not have *finite* paths that end with true nor false, by definition of $\delta : Q \times \Sigma \to \mathcal{B}^+(D \times Q)$.)

An *alternating tree automaton* $A = (\Sigma = 2^O, D = 2^I, Q, q_0, \delta, acc)$ *accepts a computation tree* $(V = (2^I)^*, l : V \to 2^O)$, written $(V, l) \models A$, iff the automaton has an accepting run-tree on that computation tree. An alternating tree automaton is *non-empty* iff there exists a computation tree accepted by it.

Similarly, *a Moore system* $M = (I, O, T, t_0, \tau, out)$ *is accepted by the alternating tree automaton* $A = (\Sigma = 2^O, D = 2^I, Q, q_0, \delta, acc)$, written $M \models A$, iff $(V, l) \models A$, where $(V = (2^I)^*, l : V \to 2^O)$ is the system computation tree.

Let us define different variations of an acceptance condition $acc : Q^\omega \to \mathbb{B}$. For a given infinite sequence $\pi \in Q^\omega$, let $Inf(\pi)$ be the elements of Q appearing in π infinitely often. Then:

- *Büchi acceptance* is defined by a set $F \subseteq Q$: $acc(\pi)$ holds iff $Inf(\pi) \cap F \neq \emptyset$. We often call the states of F *accepting*.
- *Co-Büchi acceptance* is defined by a set $F \subseteq Q$: $acc(\pi)$ holds iff $Inf(\pi) \cap F = \emptyset$. We often call the states of F *rejecting*.
- *Streett acceptance* is defined by pairs $\{(A_i \subseteq Q, G_i \subseteq Q)\}_{i \in [k]}$: $acc(\pi)$ holds iff $\forall i \in [k] : Inf(\pi) \cap A_i \neq \emptyset \to Inf(\pi) \cap G_i \neq \emptyset$.
- *Rabin acceptance* is defined by pairs $\{(F_i, I_i)\}_{i \in [k]}$: $acc(\pi)$ holds iff $\exists i \in [k] : Inf(\pi) \cap F_i = \emptyset \wedge Inf(\pi) \cap I_i \neq \emptyset$.
- *Parity acceptance* is defined by a priority function $p : Q \to [0, k]$: $acc(\pi)$ holds iff the minimal priority appearing infinitely often in $p(\pi)$ is even.

In addition to the above acceptance conditions, we define generalized versions. Generalized Büchi acceptance condition is defined by a set $\{F_i\}_{i \in [k]}$: $acc(\pi)$ holds iff the Büchi condition holds wrt. every F_i where $i \in [k]$. Similarly define Generalized co-Büchi[2], Streett, Rabin, and Parity conditions.

Nondeterministic and universal tree automata

Depending on the form of $\delta(q, \sigma)$ (for every $(q, \sigma) \in Q \times \Sigma$), we distinguish the following special cases of alternating tree automata.

- *Universal* tree automata: $\delta(q, \sigma)$ is a conjunction of variables of $Q \times D$, where each direction is mentioned at least once.
- *Deterministic* tree automata: $\delta(q, \sigma)$ is a conjunction of variables of $Q \times D$ and each direction is mentioned exactly once.
- *Nondeterministic* tree automata: let $\delta(q, \sigma)$ be rewritten in DNF. Then each conjunct mentions each direction exactly once.

[2]We stress that, in our work, Generalized co-Büchi for a set $\{F_i\}_{i \in [k]}$ means: $acc(\pi)$ holds iff the co-Büchi condition holds wrt. *every* F_i where $i \in [k]$. But often Generalized co-Büchi acceptance means that *there exists* F_i that is visited finitely often where $i \in [k]$.

Alternating hesitant tree automata (AHT)

An *alternating hesitant tree automaton (AHT)* is an alternating tree automaton $(\Sigma, D, Q, q_0, \delta, acc)$ with the following acceptance condition and structural restrictions. The restrictions reflect the fact that AHTs are tailored for CTL* formulas.

- Q can be partitioned into $Q_1^N, \ldots, Q_{k_N}^N$, $Q_1^U, \ldots, Q_{k_U}^U$, where superscript N means nondeterministic and U means universal. Let $Q^N = \bigcup Q_i^N$ and $Q^U = \bigcup Q_i^U$. (Intuitively, nondeterministic state sets describe E-quantified subformulas of the CTL* formula, while universal state sets describe A-quantified subformulas.)
- There is a partial order on $\{Q_1^N, \ldots, Q_{k_N}^N, Q_1^U, \ldots, Q_{k_U}^U\}$. (Intuitively, this is because state subformulas can be ordered according to their relative nesting.)
- The transition function δ satisfies: for every $q \in Q$, $a \in \Sigma$
  - if $q \in Q_i^N$, then: $\delta(q, a)$ contains only disjunctively related[1] elements of Q_i^N; every element of $\delta(q, a)$ outside of Q_i^N belongs to a lower set;
  - if $q \in Q_i^U$, then: $\delta(q, a)$ contains only conjunctively related[1] elements of Q_i^U; every element of $\delta(q, a)$ outside of Q_i^U belongs to a lower set.

Finally, $acc : Q^\omega \to \mathbb{B}$ of AHTs is defined by a set $Acc \subseteq Q$: $acc(\pi)$ holds for $\pi = q_1 q_2 ... \in Q^\omega$ iff one of the following holds.

- The sequence π eventually stays in some Q_i^U and $Inf(\pi) \cap (Acc \cap Q^U) = \emptyset$ (co-Büchi acceptance). Let us denote $F = Acc \cap Q^U$.
- The sequence π eventually stays in some Q_i^N and $Inf(\pi) \cap (Acc \cap Q^N) \neq \emptyset$ (Büchi acceptance). Let us denote $I = (Acc \cap Q^N) \cup (Q^U \setminus Acc)$.

Due to the restrictions on the structure of hesitant automata, this acceptance is equivalent to the Rabin acceptance with one pair (F, I).

2.5 Word Automata

In contrast to tree automata that consume infinite trees, word automata consume infinite words. Every LTL formula can be translated into a word automaton, but a CTL* formula, in general, cannot. This is because a CTL* formula describes a set of trees, while an LTL formula describes a set of words.

We start with alternating word automata. Such automata can concisely represent LTL formulas (without incurring an exponential blow-up in its size). Then we define two specializations: nondeterministic and universal word automata. Such automata are often used as input to synthesis algorithms, because they are simpler to work with (although translation of an LTL formula into such an automaton

[1] In a Boolean formula, atoms E are disjunctively [conjunctively] related iff the formula can be written into DNF [CNF] in such a way that each cube [clause] has at most one element from E.

can incur and exponential blow-up). Finally, we define alternating hesitant word automata. They are useful for model checking and synthesis from AHTs (and thus from CTL* formulas).

Alternating word automata

Remark 3 (Re-using definitions from tree automata)**.** An infinite word can be viewed as a tree with a single branch. Thus it is tempting to derive definitions for word automata from those of tree automata. Without additional tricks this will not work for the following reason. In our work, branching degree of every computation tree is, by definition, $|2^I|$. Thus, considering only single-branch trees is equivalent to having systems with no inputs: $|2^I| = 1 \Leftrightarrow |I| = 0$. But we are interested in the general case: $|I| \in \mathbb{N}_0$. (To reuse definitions, we could move the inputs into the outputs, consider nondeterministic systems without edge labels, and require that each state has a successor containing a label e, for every $e \in 2^I$.)

An *alternating word automaton* is a tuple $(\Sigma, Q, q_0, \delta, acc)$ where Σ is an alphabet, Q is a set of states, $q_0 \in Q$ is initial, $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is a transition function, and $acc : Q^\omega \to \mathbb{B}$ is a path acceptance condition. Note that $\delta(q) \neq \mathsf{false}$ and $\delta(q) \neq \mathsf{true}$ for every $q \in Q$, i.e., there is a successor state for every letter. These are not real limitations, they are in place to simplify definitions.

Given a word $w = a_1 a_2 ... \in \Sigma^\omega$, let $\mathit{Pref}(w) = \{\epsilon, a_1, a_1 a_2, a_1 a_2 a_3, ...\}$ denote the set of all its prefixes (including the empty prefix). Also, for a finite non-empty word $w \in \Sigma^+$, let $last(w)$ denote its last letter.

A *run-tree* of an alternating word automaton $(\Sigma, Q, q_0, \delta, acc)$ on an infinite word $w = a_1 a_2 ... \in \Sigma^\omega$ is a tree with directions Q, labels $\mathit{Pref}(w) \times Q$, nodes $V' \subseteq (\Sigma \times Q)^*$, labeling function $l' : V' \to \mathit{Pref}(w) \times Q$ such that

- $l'(\epsilon) = (\epsilon, q_0)$,
- if $v \in V'$ with $l'(v) = (p, q)$, then:
  there exists $\{q_1, ..., q_k\}$ that satisfies $\delta(q, last(p))$ and $p \cdot q_i \in V'$ for every $i \in [1, k]$.

This definition coincides with the definition of run-trees of tree automata when restricted to trees with a singe path.

A *run-tree is accepting* iff every run-tree path starting from the root is accepting. A run-tree path $v_1 v_2 ...$ is accepting iff $acc(q_1 q_2 ...)$ holds, where q_i, for every i, is an automaton state of the label $l'(v_i)$.

An *alternating word automaton* $A = (\Sigma, Q, q_0, \delta, acc)$ *accepts an infinite word* $w = a_1 a_2 ... \in \Sigma^\omega$, written $w \models A$, iff the automaton has an accepting run-tree on the word. An alternating word automaton is *non-empty* iff there exists an infinite word accepted by it.

So far we re-used definitions from Section 2.4, but now we depart.

In Section 2.3 about CTL*, we introduced two path quantifiers: A (on all paths) and E (there exists a path). Accordingly, we define $M \models \mathsf{A}(A)$ and $M \models \mathsf{E}(A)$: $M \models \mathsf{E}(A)$ iff there is a system computation accepted by the automaton; $M \models \mathsf{A}(A)$ iff every system computation is accepted by the automaton.

Remark 4 (Tree automata vs. Word automata)**.** Consider CTL* formula $\mathsf{EX}\, g \wedge \mathsf{EX}\, \neg g$, outputs $O = \{g\}$, inputs $I = \{r\}$. No alternating *word* automaton, when prefixed with E or A, can describe this language, while there is an alternating *tree* automaton for it.

Nondeterministic and universal word automata

We now look closer at nondeterministic and universal word automata. Their definitions coincide with those for tree automata when assuming the single direction, but for clarity we recall them here.

Depending on the form of $\delta(q, a)$ (for every $(q, a) \in Q \times \Sigma$), we distinguish:

- *Deterministic* word automata: $\delta(q, a)$ is a single state. There is no choice: we know exactly into which state to proceed from q when reading a. Thus, a run-tree degenerates into a line.
- *Nondeterministic* word automata: $\delta(q, a)$ is a (non-empty) disjunction of states. We reading a in state q, we choose one of the states and proceed. A run-tree degenerates into a line.
- *Universal* word automata: $\delta(q, a)$ is a conjunction of states $q_1 \wedge ... \wedge q_k$ where $k \geq 1$. Thus, when reading a we send one copy of the automaton into every $q_1 ... q_k$. A run-tree is indeed a tree.

In all the cases above, the transition function can be expressed as $\delta : Q \times \Sigma \to 2^Q \setminus \{\emptyset\}$. We often use this notation instead of $\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$.

Alternating hesitant word automata (AHW)

An *alternating hesitant word automaton* (AHW) $A = (\Sigma, Q, q_0, \delta : Q \times \Sigma \to \mathcal{B}^+(Q), Acc \subseteq Q)$ is an alternating word automaton with the similar to AHTs structural restrictions and the same as for AHTs acceptance condition.

Tree variants of word automata

We define the following tree variants, A_A and A_E, of a word automaton A. Given a nondeterministic word automaton $A = (2^{I \cup O}, Q, q_0, \delta : Q \times 2^{I \cup O} \to 2^Q, acc)$, let $A_\mathsf{E} = (2^O, D = 2^I, Q, q_0, \delta' : Q \times 2^O \to 2^{Q \times D}, acc)$ be the nondeterministic tree automaton defined in the most natural way: for every $(q, o) \in Q \times 2^O$,

$$\delta'(q, o) = \bigvee_{d \in D} \delta\big(q, (o, d)\big)\, [q' \mapsto (d, q')]\,.$$

We define A_A in the same way.

We will use A_E to talk about "the product between system M and nondeterministic *word* automaton A". Since we do not define such a product—it is defined for *tree* automata only—we use A_E instead. Similarly, when we have universal word automata, we use A_A.

A_A and A_E satisfy the following property: for every system M,

- for universal A: $M \models \mathsf{A}(A) \;\Leftrightarrow\; M \models A_{\mathsf{A}}$,
- for nondeterministic A: $M \models \mathsf{E}(A) \;\Leftrightarrow\; M \models A_{\mathsf{E}}$.

Automata Abbreviations

We use the standard three letter abbreviation for automata

$$(\{A, U, N, D\} \times \{B, C, S, P, R\} \times \{W, T\}) \cup \{AHT, AHW\}.$$

For example, NBW means Nondeterministic Büchi Word automaton, UCT means Universal co-Büchi Tree automaton, AHT means Alternating Hesitant Tree automaton.

2.6 Approaches to Model Checking

This section treats the automata-theoretic approach to CTL* and LTL model checking [63], as well as the classical bottom-up approach to CTL* model checking [32]. Let us start with the definition.

The *model checking problem* is:

Given: a Moore system M, formula Φ in some logic
Return: does $M \models \Phi$?

Depending on the logic of Φ, we have CTL* and LTL model checking problems.

We now briefly describe the automata-based solution to CTL* (and thus LTL) model checking problem introduced by Kupferman, Vardi, and Wolper [63]. The idea is to translate a given formula Φ into a tree automaton:

- into AHT, when Φ is in CTL*,
- into UCT, when Φ is in LTL.

Then we build the "product" of the system and the automaton, which will be a word automaton (an alternating one for CTL*, a universal one for LTL). Such a word automaton captures the joint behaviours of the system and the automaton for Φ. The nice property of the product is that it is empty iff the system is accepted by the tree automaton for Φ (equivalently: iff the system satisfies Φ). Thus, we reduce the model checking problem to checking the non-emptiness of a word automaton. Below we define the product.

Product of system and tree automaton

Let us provide the intuition.

A system M can be seen as a deterministic tree automaton A_M that accepts only its own computation tree. Then the question of whether the system is accepted by a given tree automaton A is equivalent to checking whether the intersection $A_M \wedge A$ is non-empty. The product of a system M and a tree automaton

A, written $M \otimes A$, can be seen as the intersection tree automaton $A_M \wedge A$, from which we remove labels and directions[3] and which we treat as a *word* automaton. The important property is that the product $M \otimes A$—a 1-letter alternating word automaton— is empty iff $A_M \wedge A$ is empty.

A *1-letter alternating hesitant word automaton (1-AHW)* is an AHW $(Q, q_0, \delta : Q \to \mathcal{B}^+(Q), Acc \subseteq Q)$, whose alphabet has only one letter (not shown in the tuple). Informally, an 1-AHW is an and-or graph of a restricted form plus a Rabin acceptance condition.

A *product* of an AHT $A{=}(2^O, 2^I, Q, q_0, \delta, Acc)$ and a system $M{=}(I, O, T, t_0, \tau, out)$, written $M \otimes A$, is a 1-AHW $(Q \times T, (q_0, t_0), \Delta, Acc')$ such that

$$Acc' = \{(q, t) \mid q \in Acc\}$$

and for every $(q, t) \in Q \times T$:

$$\Delta(q, t) = \delta(q, out(t))[(d, q') \mapsto (\tau(t, d), q')].$$

As before, $M \otimes A$ is non-empty iff there exists an infinite word accepted by it. Since $M \otimes A$ has a one-letter alphabet—let it be $\Sigma = \{l\}$—checking the non-emptiness of $M \otimes A$ means checking whether the infinite word l^ω is accepted.

Recall that the states of an AHW can be partitioned into "existential" sets $Q^N_1, ..., Q^N_{k_N}$ and "universal" sets $Q^U_1, ..., Q^U_{k_U}$. These sets are ordered, and the transition function of the AHW satisfies the restriction that ensures the following: every infinite path of the AHW gets trapped in some Q^N_i or Q^U_j.

CTL* model checking using the product

For the general case of CTL* formulas and AHTs we know the following.

Proposition 1 ([63])**.** *A system M is accepted by an AHT A iff their product $M \otimes A$ is non-empty.*

Corollary 2. *A system M satisfies CTL* formula Φ iff the product $M \otimes A_\Phi$ is non-empty, where A_Φ is an AHT for Φ.*

The 1-AHW non-emptiness problem can be decided in linear time wrt. the size of the automaton [63], therefore the approach is EXPTIME wrt. the size of a given CTL* formula. It is known [9] that CTL* model checking problem is PSPACE-complete.

LTL model checking using the product

Consider the special case of LTL properties. We can check whether a given system M satisfies an LTL formula φ as follows. First, construct a UCT A_φ for φ (we do not need alternating automata for LTL). Second, build the product $M \otimes A_\varphi$. Such a product is a 1-letter *universal* word automaton (1-UCW). Then the model checking is equivalent to checking non-emptiness of the 1-UCW.

[3]We can remove labels because, for each state of A_s, the label is uniquely defined—it is $out(t)$ for the corresponding system state t. We can remove directions, because in the non-emptiness check of a 1-letter tree automaton we do not distinguish directions.

Proposition 3. *A system M satisfies an LTL formula φ iff the product $M \otimes A_\varphi$ is non-empty, where A_φ is the UCT for φ.*

For the same arguments as for CTL*, the approach is EXPTIME wrt. the size of a given LTL formula.

Bottom-up CTL* model checking

Here is the classical construction [32] (see also [9, p.427]) for CTL* model checking.

We will need the following notions of F, P, and $\widetilde{\Phi}$. Let Φ be a CTL* formula with inputs I and outputs O. Let $F' = \{f'_1, ..., f'_k\}$ be the set of CTL* subformulas of the form $\mathsf{A}\varphi$ or $\mathsf{E}\varphi$, where φ is a path formula. The subformulas $\{f'_1, ..., f'_k\}$ can be ordered wrt. path quantifier nesting depth d. Let us assume that $f'_1, ..., f'_k$ are ordered wrt. d in increasing order, i.e., $d(f'_i) \leq d(f'_{i+1})$ for every i. With every f'_i we associate a proposition p_i, which makes up the set $P = \{p_1, ..., p_k\}$. Let $F = \{f_1, ..., f_k\}$ be the set of formulas, where each f_i is f'_i in which all subformulas were replaced by the corresponding propositions: For example, for the CTL* formula $\Phi = \mathsf{EG}(g \rightarrow g\,\mathsf{U}\,\mathsf{AX}\neg g)$ we have $F' = \{f'_1 = \mathsf{AX}\neg g, f'_2 = \mathsf{EG}(g \rightarrow g\,\mathsf{U}\,\mathsf{AX}\neg g)\}$, $P = \{p_1, p_2\}$, and $F = \{f_1 = \mathsf{AX}\neg g, f_2 = \mathsf{EG}(g \rightarrow g\,\mathsf{U}\,p_1)\}$. Notice about F: (i) F are formulas over atoms $I \cup O \cup P$, (ii) every f_i is over terms $I \cup O \cup \{p_1, ..., p_{j \leq i-1}\}$, and (iii) they are of the form $\mathsf{A}\varphi$ or $\mathsf{E}\varphi$, where φ is a CTL* path formula that has *no path quantifiers*. Let $\widetilde{\Phi}$ be Φ where all subformulas were replaced by the corresponding propositions. Note that $\widetilde{\Phi}$ is a *Boolean* formula over $O \cup P$.

Given a CTL* formula Φ and a system M. The bottom up model checker creates the formulas F, propositions P, and $\widetilde{\Phi}$. Then it annotates the system states with propositions from P such that a proposition p_i holds in a state t iff $t \models f_i$. It does so in a bottom up manner (inductively):

- It starts with the proposition p_1: the formula f_1 is a path formula over propositions $I \cup O$. We can use LTL model checker to check if $t \models f_1$, for every system state t.
- Similarly for p_i: use LTL model checker to check if $t \models f_i$, where f_i can talk about propositions $I \cup O \cup \{p_1, ..., p_{i-1}\}$ whose truth for every system state we already established.
- Finally, $M \models \Phi$ iff $\widetilde{\Phi}$ holds in the initial system state.

The complexity of the procedure is EXPTIME wrt. $|\Phi|$.

2.7 Approaches to Synthesis

This section describes (i) the classical game-based approach [71] to LTL synthesis, (ii) a more recent approach [46] that avoids automata determinization and uses constraint solvers, and (iii) an approach to CTL* synthesis. Let us start with the definition.

The *synthesis problem* is:

Given: the set of inputs I, the set of outputs O, formula Φ in some logic
Return: a Moore system with inputs I and outputs O satisfying Φ,
or "unrealisable" if no such system exists

The input $\langle I, O, \Phi \rangle$ to the problem is called a *specification*. A specification is *realisable* if the answer is a system, otherwise the specification is *unrealisable*. Depending on the logic of Φ, we have LTL and CTL* synthesis problems. Instead of a *formula* Φ, we can use tree automata or word automata prefixed with the E or A path quantifier.

It is known [62, 71] that the CTL* and LTL synthesis problems are 2EXPTIME-complete. Below we discuss two approaches to LTL synthesis problem, game-based approach and bounded synthesis. Then we discuss an approach to CTL* synthesis.

LTL synthesis via reduction to games

The standard game-based approach [71] to synthesis from LTL specifications is as follows.

- Translate a given LTL formula into a nondeterministic Büchi word automaton [82]. The automaton can be exponentially larger than the LTL formula.
- Determinise the automaton into a deterministic parity word automaton, e.g. using Safra construction [77]. The resulting automaton can be exponentially larger than the original one, leading to the doubly exponential blow up.
- Translate the word automaton into a tree automaton, by splitting each transition into two transitions according to the input and output labels.
- Check the non-emptiness of the deterministic parity tree automaton. The check can be done by treating the tree automaton as a game, and then solving the game. If the game is winning for the system player (it controls the choice of output labels), then the specification is realisable, otherwise it is unrealisable. The particular class of parity games that we get can be solved in polynomial time e.g. using [26].

The method gives 2EXPTIME solution to the synthesis problem.

The 2EXPTIME-hardness comes from the fact that we can encode into the LTL realisability problem the acceptance of a given word by an alternating exponential-space bounded Turing machine [72, 80].[4] I.e., given such a Turing machine and an input word, we can build the LTL specification, which is realisable iff the word is accepted by the machine. The length of the specification is 2^{cn} where n is the length of the input word and c is a constant. The specification requires a system to output, in each step, the set of all successor configurations of the TM until it accepts on all of them. In each step, with the aid of additional inputs, we choose

[4] An alternating Turing machine is, like an alternating automaton, has universal and existential transitions. A given word is accepted if there is an accepting run-tree of the machine on this word (and thus all its branches are accepting). A Turing machine is exponential-space bounded iff: (i) it terminates on all inputs, (ii) it uses 2^{cn} number of cells where n is the length of the input word and c is a constant. The problem of deciding whether a word is accepted by such a machine is 2EXPTIME-complete [27].

one configuration from which to proceed. Non-deterministic transitions of the TM are emulated using ORs in the LTL formula.

Bounded LTL synthesis via SMT

The idea of bounded synthesis via SMT [46] is to reduce the synthesis problem to SMT solving. The resulting SMT query encodes the *model checking* question—the query is satisfiable iff the system satisfies a given specification. To turn model checking into synthesis, we replace the given system by uninterpreted functions. Therefore, if the query is satisfiable, then the SMT solver produces—in addition to YES/NO answer—models of the uninterpreted functions that encode the system. From those models we extract the system, and such a system is correct.

The SMT query encodes the non-emptiness of the product of a system and a UCT, where UCT represents a given LTL formula (see also Proposition 3 on page 22). Recall that such a product is a 1-UCW. Thus, the emptiness question reduces to finding a lasso with a final state of the 1-UCW in the loop.

If a system was given (as in the model checking), then using SMT solvers in *this* way to solve such a simple graph question does not seem[5] to be wise (if we fix a system, then the complexity of solving such[6] SMT queries is NPTIME-hard wrt. the size of the formula automaton, while the straight graph-based approach is in PTIME wrt. the size of the formula automaton; also, the SMT-based approach is not symbolic). But in the case of synthesis the system is not given: here, an SMT solver plays the role of an efficient guess-verify searcher.

The pseudo-code of the bounded synthesis is:

```
convert a given LTL formula into UCT
for system size in {1...bound}:
    encode non-emptiness of system*UCT into SMT query
    solve the query
    if the query is satisfiable:
        return REALIZABLE
return UNREALIZABLE
```

Let us go through the steps.

Automata translation. A given LTL formula φ is translated into a UCT U which accepts a Moore machine M iff M satisfies φ. This can be done, for example, using SPOT [35] or LTL3BA [8]: negate the formula, translate it into a NBW, treat it as a UCW A, and turn the UCW A into a UCT A_{A} as described on page 19. This A_{A} is the sought UCT U.

Iteration for increasing bounds. Fix the number of states in a system M. This allows us to encode the non-emptiness problem of $M \otimes U$ into a decidable fragment of SMT. The bound *bound* can be either user-chosen or it is the upper

[5] SMT solvers *are* used in verification, see, for example, papers on solving Horn clauses [14, 11, 12]. Here we refer only to the way of using them as it is done in bounded synthesis.

[6] Here "such queries" means that they have the same theory as those used by bounded synthesis (for example, UFLIA). We did not analyse whether the special structure of SMT queries from bounded synthesis gives way to a simpler complexity than that of solving general UFLIA queries.

bound on the system size ($O(2^{2^{|\varphi|}})$). A better way, from the practical point of view, is described in Remark 5.

SMT encoding. Let inputs and outputs be I and O. Fix the states T of a system $M = (I, O, T, t_0, \tau, out)$. Let UCT $U = (2^O, 2^I, Q, q_0, \delta' : Q \times 2^O \to 2^{Q \times 2^I}, F \subseteq Q)$. In the SMT query, we use uninterpreted functions to express system functions δ and out. We also use two uninterpreted functions: $rch : Q \times T \to \mathbb{B}$ denotes whether a pair $(q, t) \in Q \times T$ is reachable in the product $M \otimes U$, and $\rho : Q \times T \to \mathbb{N}$ which is called *ranking function* and is used to ensure the absence of bad lassos (they visit $q \in F$ in the loop of the lasso). The constraints are:

$$rch(q_0, t_0) \wedge \bigwedge_{(q,t) \in Q \times T} \Big[rch(q,t) \to \bigwedge_{(d,q') \in \delta(q, out(t))} rch(q', \tau(t,d)) \;\wedge\; \rho(q,t) \;\triangleright\; \rho(q', \tau(t,d)) \Big] \tag{2.1}$$

where $\triangleright$ is $>$ if $q \in F$, otherwise $\geq$. The intuition is as follows. We mark the initial state (q_0, t_0) of the product $M \otimes U$ as reachable. For every reachable state (q, t), we mark every successor state $(q', \tau(t, d))$ of the product as reachable, and we require the rank to strictly decrease if $q \in F$, and non-strictly decrease otherwise. Thus, all reachable states of the product are marked with rch. Additionally, if there is a bad lasso (that has (q_b, t) with $q_b \in F$ inside its loop), then the query will have an unsatisfiable cycle of constraints $(q_b, t) > ... \geq (q_b, t)$. Note that this query is satisfiable iff there exists functions τ and out (and ρ and rch) such that the product does no have a bad lasso [46].

Solving the SMT query. To solve the query one can use e.g. Z3 solver [34].

Remark 5 (Checking unrealisability)**.** When a given LTL specification is unrealisable, the above procedure iterates through all system sizes up to a bound $O(2^{2^{|\varphi|}})$. The bound is computationally difficult to reach on non-toy unrealisable specifications, making the approach impractical. To overcome this, we can use the determinacy of the LTL synthesis problem, which states: an LTL specification is unrealisable iff the dual LTL specification is realisable. For a specification $\langle I, O, \varphi, \mathit{Moore} \rangle$ the specification $\langle O, I, \neg\varphi, \mathit{Mealy} \rangle$ is called dual, i.e., we swap inputs and outputs, negate the formula, and search for a Mealy machine instead of a Moore machine. (Mealy machines are just like Moore machines except that the output function $out : T \times 2^I \to 2^O$ also depends on inputs.) Thus, instead of iterating for increasing system bound, we can run two processes in parallel: one checks for realisability of the original specification, another checks for realisability of the dual specification. The process that finishes first, returns the answer, while the other process is terminated. This approach is used in most bounded synthesis implementations [1].

CTL* synthesis

The standard approach to CTL* synthesis [62] is: translate a given CTL* formula Φ into an alternating Rabin tree automaton [63] with $\approx 2^{|\Phi|}$ many states and $\approx |\Phi|$ many acceptance pairs, turn it into a nondeterministic Rabin tree automaton [67]

with $\approx 2^{2^{|\Phi|}}$ many states and $\approx 2^{|\Phi|}$ many acceptance pairs, and check its non-emptiness. The latter check is polynomial in the size of the automaton [42, 71], i.e., requires $\approx 2^{2^{|\Phi|}}$ time. Thus, the approach gives a 2EXPTIME algorithm. The lower bound comes from the 2EXPTIME completeness of the LTL synthesis problem [72, 80] and the fact that CTL* subsumes LTL.

Chapter 3

Bounded Synthesis for Streett, Rabin, and CTL*

This chapter is based on joint work with Roderick Bloem [56].

Abstract. SMT-based Bounded Synthesis uses an SMT solver to synthesize systems from LTL properties by going through co-Büchi automata. In this chapter, we show how to extend the ranking functions used in Bounded Synthesis, and thus the bounded synthesis approach, to Büchi, Parity, Rabin, and Streett conditions. We show that we can handle both existential and universal properties this way, and therefore, that we can extend Bounded Synthesis to CTL*. Thus, we obtain the first Safraless synthesis approach and the first synthesis tool for (conjunctions of) the acceptance conditions mentioned above, and for CTL*.

3.1 Introduction

For Linear Temporal Logic [70], the standard approach to reactive synthesis involves Safra's relatively complex construction [77] to determinize Büchi automata [71]. The difficulty to implement the construction has led to the development of *Safraless* approaches [61, 46]. In this chapter, we focus on one such approach, called Bounded Synthesis, introduced by Finkbeiner and Schewe [46].

The idea behind Bounded Synthesis is the following. LTL properties can be translated to Büchi automata [82] and verification of LTL properties can be reduced to deciding emptiness of the product of this automaton and the Kripke structure representing a system [66, 83] (see also Section 2.7). This product is a Büchi automaton in its own right. Finkbeiner and Schewe made two important observations: (1) Using a ranking function, the emptiness problem of Büchi automata can be encoded as a Satisfiability modulo Theories (SMT) query, and (2) by fixing its size, the Kripke structure can be left uninterpreted, resulting in an SMT query for a system that fulfills the property. Because the size of the system is bounded by Safra's construction, this yields an approach to LTL synthesis that is complete in principle. (Although proofs of unrealizability are usually computed differently.)

The reduction to SMT used by Bounded Synthesis provides two benefits: the performance progress of SMT solvers and the flexibility. The flexibility allows one to easily adapt the SMT constraints, produced by Bounded Synthesis, to build semi-complete synthesizers for distributed [46], self-stabilising [16], parameterized [49], assume-guarantee [17], probabilistic [10], and partially implemented systems.

In this chapter, we extend Bounded Synthesis in two directions.

First, we show how to directly encode into SMT that some path of a system is accepted by an X automaton, for $X \in \{\text{Büchi, co-Büchi, Parity, Streett, Rabin}\}$. We do this by introducing new ranking functions. Therefore we avoid the explicit translation of these automata into Büchi automata, which would be needed if we were to use the original Bounded Synthesis.

Second, we extend Bounded Synthesis to the branching logic CTL*. CTL* formulas allow the user to specify *structural* properties of the system. For example, if g is system output and r is system input, then the CTL* formula $\mathsf{AG}\,\mathsf{EF}\,g$ says that a state satisfying g is always reachable; and the CTL* formula $\mathsf{EFG}(g \wedge r)$ says that a state satisfying g is reachable and it has a loop when reading r that satisfies g. In both cases, the existential path quantifier E allows us to refrain from specifying the exact path that leads to such states.

In this chapter we show *two* Bounded Synthesis approaches for CTL*. First, we show how to use the ranking functions for X automata to either decide that some path of a system fulfills such a condition, or that *all* paths of the system do. Once we have established this fact, we can extend Bounded Synthesis to logics like CTL* by replacing all state subformulas by fresh atomic propositions and encoding them each by a Büchi automaton. This approach follows the classical construction [32] of model checking CTL*, extending it to synthesis setting. Alternatively, we show that we can use a translation of CTL* to Alternating Hesitant Tree Automata [63] to obtain a relatively simple encoding to SMT.

Thus, we obtain a relatively simple, Safraless synthesis procedure to (conjunctions of) various acceptance conditions and CTL*. This gives us a decision procedure that is efficient when the specification is satisfied by a small system, but is admittedly impractical at showing unrealizability. Just like Bounded Synthesis does for LTL synthesis, it also gives us a semi-decision procedure for undecidable problems such as distributed [73] or parameterized synthesis [49, 57]. We have implemented the CTL* synthesis approach in a tool[1] that to our knowledge is the only tool that supports CTL* synthesis.

The chapter is structured as follows. In the next section we list the definitions that this chapter uses. Then in Section 3.2 we introduce ranking functions that can be used to verify and synthesize properties expressed as word automata. Section 3.3 contains two approaches to Bounded Synthesis for CTL*: Section 3.3.1 describes the direct encoding into SMT, in the spirit of bottom-up CTL* model checking, while Section 3.3.2 describes the approach via hesitant tree automata. Section 3.3.3 describes the prototype CTL* synthesizer and the experiments that show applicability of the approach for the synthesis of small monolithic and distributed systems.

[1] Available at https://github.com/5nizza/party-elli, branch "cav17".

3.2 Synthesis from Büchi, Streett, Rabin, and Parity Automata

In this section we describe how to verify and synthesize properties described by Büchi, co-Büchi, Parity, Streett, and Rabin conditions. For each acceptance condition $X \in \{\text{Büchi, co-Büchi, Parity, Streett, Rabin}\}$, we can handle the question whether (the word defined by) some path of a system is in the language of a nondeterministic X automata, as well as the question of whether all paths of the system are in the language defined by a universal X automaton. There does not appear to be an easy way to mix these queries ("do all paths of the system fulfill the property defined by a given nondeterministic automaton?").

3.2.1 Preliminaries on Ranking

In the following, given a system $M = (I, O, T, t_0, \tau, out)$ and a nondeterministic (universal) word automaton $A = (2^{I \cup O}, Q, q_0, \delta, acc)$, we describe how to build an SMT query $\Phi(M, A)$ that is satisfiable iff some path (all paths, resp.) of M are in $L(A)$. That is, we focus on the verification problem. When the verification problem is solved, we obtain the solution to the synthesis problem easily, following the Bounded Synthesis approach: given an automaton A, we ask the SMT solver whether there is a system M such that $\Phi(M, A)$ is satisfiable. More precisely, for increasing k, we fix a set of k states and ask the SMT solver for a transition relation τ and a labeling out (and a few more objects) for which $\Phi(M, A)$ is satisfiable.

Our constructions use ranking functions. A *ranking function* is a function $\rho : Q \times T \to D$ for some totally ordered set D with order $\geq$. A *rank comparison relation* is a (possibly partial) relation $\triangleright \subseteq Q \times D \times D$. In the following, we write $d \triangleright_q d'$ to mean $(q, d, d') \in \triangleright$. We will usually define $\triangleright$ using $\geq$ and ρ.

We will first establish how to use the ranking functions to check existential and universal properties, expressed as $\mathsf{E}(A)$ and $\mathsf{A}(A)$. Then we define the ranking functions and comparisions for the different acceptance conditions, i.e., for different types of the word automaton A.

Given a rank comparison $\triangleright$, we define the following formula to check an existential property $\mathsf{E}(A)$:

$$\Phi^{\triangleright}_{\mathsf{E}}(M, A) = rch(q_0, t_0) \wedge \bigwedge_{q,t \in Q \times T} rch(q, t) \to \bigvee_{(q, i \cup o, q') \in \delta} out(t) = o \wedge rch(q', \tau(t, i)) \wedge \rho(q, t) \triangleright_q \rho(q', \tau(t, i)).$$

Similarly, to check a universal property $\mathsf{A}(A)$, we define

$$\Phi^{\triangleright}_{\mathsf{A}}(M, A) = rch(q_0, t_0) \wedge \bigwedge_{q,t \in Q \times T} rch(q, t) \to \bigwedge_{(q, i \cup o, q') \in \delta} out(t) = o \to rch(q', \tau(t, i)) \wedge \rho(q, t) \triangleright_q \rho(q', \tau(t, i)).$$

In these formulas,

- the free variable $rch : Q \times T \to \mathbb{B}$ is an uninterpreted function that marks reachable states in the product of M and A_{E} or A_{A}, where A_{E} and A_{A} are the tree automata for $\mathsf{E}(A)$ and $\mathsf{A}(A)$ (defined on page 19), and
- the free variable $\rho : Q \times T \to D$ is an uninterpreted ranking function.

Intuitively, $\Phi^{\triangleright}_{\mathsf{E}}$ will be used to encode that there is an accepting loop in the product automaton, while $\Phi^{\triangleright}_{\mathsf{A}}$ will be used to ensure that all loops are accepting.

Given a path $\pi = (q_1, t_1)(q_2, t_2) \cdots \in (Q \times T)^\omega$, a rank comparison relation $\triangleright$, totally ordered set D, and a ranking function ρ, π *satisfies* $\triangleright$ *using* ρ *and* D, written $(\pi, D, \rho) \models \triangleright$, iff $\rho(q_i, t_i) \triangleright_q \rho(q_{i+1}, t_{i+1})$ holds for every i.

Let us look at the properties of these equations.

Lemma 4. *For every totally ordered set D, rank comparison relation $\triangleright$, ranking function ρ, nondeterministic word automaton A, and machine M: $\Phi^{\triangleright}_{\mathsf{E}}(M, A)$ is satisfiable using ρ and D iff the product $M \otimes A_{\mathsf{E}}$ has an infinite path that satisfies $\triangleright$ using ρ and D.*

Proof idea. Direction $\Leftarrow$. Let us assume that the product contains a path $\pi = (q_1, t_1)(q_1, t_1) \ldots$ such that $(\pi, D, \rho) \models \triangleright$. By definition, $\rho(q_i, t_i) \triangleright \rho(q_{i+1}, t_{i+1})$ holds for every i. If we set $rch(q, t)$ to true for $(q, t) \in \pi$ and to false for all the other states, then the formula $\Phi^{\triangleright}_{\mathsf{E}}(M, A)$ holds.

Direction $\Rightarrow$. Let us assume that $\Phi^{\triangleright}_{\mathsf{E}}$ is satisfiable, then there is a model for rch. We can use rch to construct a lasso-shaped infinite path π such that $(\pi, D, \rho) \models \triangleright$ and that belongs to the product. □

A similar result holds for universal word automata.

Lemma 5. *For every well-founded domain D, rank comparison relation $\triangleright$, ranking function ρ, universal word automaton A, and machine M: $\Phi^{\triangleright}_{\mathsf{A}}(M, A)$ is satisfiable using ρ and D iff in the product $M \otimes A_{\mathsf{A}}$ every infinite path satisfies $\triangleright$ using ρ and D.*

Proof idea. Direction $\Leftarrow$. If we set rch to true for every (q, t) reachable in the product $M \otimes A_{\mathsf{A}}$, then $\Phi^{\triangleright}_{\mathsf{A}}(M, A)$ holds.

Direction $\Rightarrow$. (Note that rch may mark some (q, t) with true, although it is not reachable in the product $M \otimes A_{\mathsf{A}}$. But for any reachable (q, t), $rch(q, t)$ holds.) We prove this direction by contradiction. Assume that there is an infinite path $\pi = (q_1, t_1)(q_2, t_2) \ldots$ such that $(\pi, D, \rho) \not\models \triangleright$. Hence there is i such that $\neg\big(\rho(q_i, t_i) \triangleright \rho(q_{i+1}, t_{i+1})\big)$. Since (q_i, t_i) is reachable (thus $rch(q_i, t_i) = \mathsf{true}$) and $M \otimes A_{\mathsf{A}}$ has a transition into (q_{i+1}, t_{i+1}), this falsifies $\Phi^{\triangleright}_{\mathsf{A}}(M, A)$ when using ρ. Contradiction. □

These two lemmas will help us to establish the main results: $M \models A_{\mathsf{E}}$ whenever $\Phi^{\triangleright}_{\mathsf{E}}(M, A)$ is satisfiable, and $M \models A_{\mathsf{A}}$ whenever $\Phi^{\triangleright}_{\mathsf{A}}(M, A)$ is satisfiable, where the word automata A are nondeterministic and universal respectively, with different acceptance conditions, and the form of $\triangleright$ in $\Phi^{\triangleright}_{\mathsf{E}}$ and $\Phi^{\triangleright}_{\mathsf{A}}$ depends on the acceptance condition. In the next sections, we describe rank comparison relations $\triangleright$ for the acceptance conditions Büchi, co-Büchi, Streett, Rabin, and Parity. For didactic purposes, let us start with the relatively simple Büchi and co-Büchi conditions.

3.2.2 Ranking for Büchi Automata

Büchi conditions were also presented in [18] and implicitly in [10]. Given a Büchi automaton $A = (2^{I \cup O}, Q, q_0, \delta, F)$, we define the rank comparison relation $\triangleright_B^A$ as

$$\rho(q,t) \triangleright_B^A \rho(q',t') = \begin{cases} \mathsf{true} & \text{if } q \in F, \\ \rho(q,t) > \rho(q',t') & \text{if } q \notin F. \end{cases} \tag{3.1}$$

Theorem 6 ([18, 10])**.** *Let D be $\mathbb{N}_0$. For every universal Büchi word automaton U, nondeterministic Büchi word automaton N, and machine M:*

- $M \models \mathsf{E}(N)$ *iff* $\Phi_{\mathsf{E}}^{\triangleright}(M,N)$ *is satisfiable, where* $\triangleright = \triangleright_B^N$.
- $M \models \mathsf{A}(U)$ *iff* $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$ *is satisfiable, where* $\triangleright = \triangleright_B^U$.

Proof idea. Consider the first item, direction $\Leftarrow$. If $\Phi_{\mathsf{E}}^{\triangleright}(M,N)$ is satisfiable, then, using the model of *rch*, we can extract a lasso-shaped path $\pi = (q_1,t_1)(q_2,t_2)\dots$ of $M \otimes N_{\mathsf{E}}$, which satisfies $\triangleright$ for every i. Such a path visits at least one accepting state of N in its loop part and therefore is Büchi accepting.

Consider the direction $\Rightarrow$. There is an infinite path of $M \otimes N$, in the shape of a lasso, that has an accepting state in its loop. We set $rch(q,t) = \mathsf{true}$ for every state (q,t) visited on the lasso-path, and set $\rho(q,t)$ to the shortest distance to an accepting state. Such *rch* and ρ make $\Phi_{\mathsf{E}}^{\triangleright}(M,N)$ hold.

Consider now the case $M \models \mathsf{A}(U)$. The direction $\Leftarrow$ is simple, consider the direction $\Rightarrow$. We describe ρ and *rch* that make $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$ hold. For every (q,t) reachable in $M \otimes U_{\mathsf{A}}$, let $rch(q,t) = \mathsf{true}$. For every reachable (q,t), let $\rho(q,t)$ be a longest distance to an accepting state. These ρ and *rch* make $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$ hold. □

Note that in the theorem a machine M is either fixed (then we solve the model checking problem), or we fix the number of states in M and express it using uninterpreted functions (then we solve the bounded synthesis problem). Also note that we used the set $\mathbb{N}$ of natural numbers for D, but we could prove the results for some other large-enough well-founded sets.

3.2.3 Ranking for co-Büchi Automata

This case was presented in the original paper [46] on Bounded Synthesis. Given a co-Büchi automaton $A = (2^{I \cup O}, Q, q_0, \delta, F)$, the ranking constraint relation $\triangleright_C^A$ for co-Büchi is defined as

$$\rho(q,t) \triangleright_C^A \rho(q',t') = \begin{cases} \rho(q,t) > \rho(q',t) & \text{if } q \in F, \\ \rho(q,t) \geq \rho(q',t') & \text{if } q \notin F. \end{cases} \tag{3.2}$$

Theorem 7 ([46])**.** *Let D be $\mathbb{N}_0$. For every universal co-Büchi word automaton U, nondeterministic co-Büchi word automaton N, and machine M:*

- $M \models \mathsf{E}(N)$ *iff* $\Phi_{\mathsf{E}}^{\triangleright}(M,N)$ *is satisfiable, where* $\triangleright = \triangleright_C^N$.
- $M \models \mathsf{A}(U)$ *iff* $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$ *is satisfiable, where* $\triangleright = \triangleright_C^U$.

Proof idea. Consider the first item. Direction $\Rightarrow$: $M \otimes N_{\mathsf{E}}$ has an infinite path, in the shape of a lasso, that never visits a rejecting state in the loop. We set $rch(q,t) = \mathsf{true}$ for all reachable (q,t) in the path, and set $\rho(q,t)$ to be the number of rejecting states visited before entering the loop. Direction $\Leftarrow$: From the model of rch we can construct an infinite path that is accepted by N. Any such path must be accepting, because having a rejecting state (q,t) visited infinitely often implies having an unsatisfiable cycle of constraints $(q,t) > ... \geq (q,t)$.

Consider the case $M \models \mathsf{A}(U)$, direction $\Rightarrow$. We set $rch(q,t) = \mathsf{true}$ for every reachable (q,t) in $M \otimes U_{\mathsf{A}}$, and set $\rho(q,t)$ to the maximal number of visits to rejecting states among the paths starting from (q,t). Such a number is finite, because all paths visit a rejecting state only finitely often. The direction $\Leftarrow$ holds, because every rejecting path visits a rejecting state (q,t) infinitely often, which implies having an unsatisfiable cycle of constraints $(q,t) > ... \geq (q,t)$. □

3.2.4 Ranking for Streett Automata

The ranking below is our contribution.

Fix a Streett automaton $A = (2^{I \cup O}, Q, q_0, \delta, \{(A_i, G_i)\}_{i \in [k]})$. We slightly modify the definitions to have $\rho : Q \times T \to D^k$ and $\triangleright \subseteq Q \times D^k \times D^k$, i.e., the ranking function consists of k components.

The ranking function $\rho : Q \times T \to D^k$ is defined using k components $\rho_i : Q \times T \to D$, $\rho(q,t) = \big(\rho_1(q,t), ..., \rho_k(q,t)\big)$. The rank comparison relation $\triangleright_S : Q \times D^k \times D^k$ is $\rho(q,t) \triangleright_S^A \rho(q',t') = \bigwedge_{i \in [k]} \Big(\rho_i(q,t) \triangleright_S^{A,i} \rho_i(q',t')\Big)$, where

$$\rho_i(q,t) \triangleright_S^{A,i} \rho_i(q',t') = \begin{cases} \mathsf{true} & \text{if } q \in G_i, \\ \rho_i(q,t) > \rho_i(q',t') & \text{if } q \in A_i \wedge q \notin G_i, \\ \rho_i(q,t) \geq \rho_i(q',t') & \text{if } q \notin A_i \cup G_i. \end{cases} \tag{3.3}$$

Theorem 8. *Let D be $\mathbb{N}_0$. For every universal Streett word automaton U, nondeterministic Streett word automaton N, and machine M:*

- $M \models \mathsf{E}(N)$ *iff* $\Phi_{\mathsf{E}}^{\triangleright}(M,N)$ *is satisfiable, where* $\triangleright = \triangleright_S^N$.
- $M \models \mathsf{A}(U)$ *iff* $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$ *is satisfiable, where* $\triangleright = \triangleright_S^U$.

Proof idea. We prove only the second item, the first item can be proven similarly.

Direction $\Rightarrow$. We construct $\rho = (\rho_1, ..., \rho_k)$ and rch that satisfy $\Phi_{\mathsf{A}}^{\triangleright}(M,U)$. Set $rch(q,t) = \mathsf{true}$ for all reachable (q,t) in the product $\Gamma = M \otimes U_{\mathsf{A}}$, and for unreachable states set rch to false and set $\rho = (0, ..., 0)$. Now let us remove all unreachable states from Γ. Then for each $i \in [k]$, ρ_i is defined as follows.

- For every $(q,t) \in \cup_{i \in [k]} G_i \times T$, let $\rho_i(q,t) = 0$.
- Define an *SCC S of a graph* to be any maximal subset of the graph states such that for any $s \in S$, $s' \in S$, the graph has a path $\pi = s, ..., s'$ of length ≥ 2, where the length is the number of states appearing on the path. Thus, a single-state SCC can appear only if the state has a self-loop.

- Remove all outgoing edges from every state (q,t) of Γ with $q \in G_i$. The resulting graph Γ' has no SCCs that have a state (q,t) with $q \in \cup_{i\in[k]} A_i$.
- Let us define the graph Γ''. Let $\mathcal{S}$ be the set of all SCCs of Γ'. Then Γ'' has the states $V_{\Gamma''} = \mathcal{S} \cup \{\{s\} \mid s \notin \cup_{S\in\mathcal{S}} S\}$, i.e., each state is either an SCC or a singleton-set containing a state outside of any SCC (but in both cases, a state of Γ' is a set of states of Γ). The edges $E_{\Gamma''}$ of Γ'' are: $(S_1, S_2) \in E_{\Gamma''}$ iff $\exists s_1 \in S_1, s_2 \in S_2 : S_1 \neq S_2 \wedge (s_1, s_2) \in E_{\Gamma'}$. Intuitively, Γ'' is a graph derived from Γ by turning all accepting states into leafs, and by making SCCs the new states. Note that the graph Γ'' is a DAG.
- Given a path $\pi = S_1, ..., S_m$ in Γ'', let $nb(\pi)$ be the number of "bad" states visited on the path, i.e., $nb = |\pi \cap \{\{(q,t)\} : q \in \cup_{i\in[k]} A_i\}|$. Such a number exists since all paths of Γ'' are finite.
- For all $(q,t) \in S \in V_{\Gamma''}$ with $q \notin G_i$, let $\rho_i(q,t)$ be the max number of "bad" states visited on any path from S: $\rho_i(q,t) = max(\{nb(\pi) \mid \pi \textit{ is a path from } S\})$. Such a number exists since the number of paths in Γ'' is finite.

This concludes the direction $\Rightarrow$.

The direction $\Leftarrow$ is proven by contradiction. Suppose $\Phi^{\triangleright}_{\mathsf{A}}(M,U)$ is satisfiable with some *rch* and $\rho = (\rho_1, ..., \rho_k)$, but $M \otimes U_{\mathsf{A}}$ is empty. The latter means that there is a lasso-shaped path that is not accepted by some pair (A_i, G_i): it visits A_i infinitely often but visits G_i only finitely often. Thus, the loop part of the path contains state (q,t) with $q \in A_i$ but has no states visiting G_i. Recall that such a path is labeled true by *rch*, because *rch* over-approximates the set of reachable states. Altogether this makes $\Phi^{\triangleright}_{\mathsf{A}}(M,U)$ unsatisfiable, because of the unsatisfiable cycle of constraints $\rho_i(q,t) > ... \geq \rho(q,t)$. □

Remark 6 (Comparison with ranking from [69])**.** Piterman and Pnueli [69] introduced ranking functions to solve Streett *games*. Our ranking functions can be adapted to solve games, too. (Recall that our SMT encoding describes *model checking* with an uninterpreted system.) It may seem that in the case of games, our construction uses fewer counters than [69], but that is not the case. Given a DSW with k Streett pairs and n states, a winning strategy in the corresponding Streett game may require a memory of size $k!$. In this case, the size of the product system×automaton is $k!n$. Our construction introduces $2k$ counters with the domain $[k!n] \to [k!n]$ to associate a rank with each state. In contrast, [69] introduces $k!k$ counters with the domain $[n] \to [0,n]$. Encoding these counters into SAT would require $2k \cdot k!n \cdot log_2(k!n)$ bits for our construction, and $k!k \cdot n \cdot log_2(n)$ bits for the construction of [69]. Thus, our construction introduces $2(1+\frac{log_2(k!)}{log_2(n)}) \approx 2(1+log_2(log_2(n)))$ times more bits (the approximation assumes that $k = log_2(n)$ and n is large). On the positive side, our construction is much simpler.

Ranking for Parity Automata

Given a Parity automaton $A = (2^{I\cup O}, Q, q_0, \delta, p)$ with priorities $0, \ldots, k-1$, it is known that we can translate it into an equivalent Streett automaton with pairs

$(A_1, G_1), \ldots, (A_{m/2}, G_{m/2})$, where $A_i = \{q \mid p(q) = 2i-1\}$, $G_i = \{q \mid p(q) \in \{0, 2, \ldots, 2i-2\}\}$. We can then apply the encoding for Streett automata. The resulting ranking resembles Jurdziński's progress measure [52].

3.2.5 Ranking for Rabin Automata

Given a Rabin automaton $A = (2^{I \cup O}, Q, q_0, \delta, \{(F_i, I_i)\}_{i \in [k]})$ and a system $M = (I, O, T, t_0, \tau, out)$, we use ranking constraints described by Piterman and Pnueli [69] to construct a rank comparison relation. The ranking function $\rho : Q \times T \to \mathbb{N}_0^{2k+1}$ maps a state of the product to a tuple of numbers $(b,\ j_1{,}d_1, \ldots, j_k{,}d_k)$, where the numbers have the following meaning. For each $l \in [k]$,

- $j_l \in [k]$ is the index of a Rabin pair,
- $b \in [0, |Q \times T|]$ is an upper bound on the number of times the set F_{j_1} can be visited from (q, t),
- $d_l \in [0, |Q \times T|]$ is the maximal distance from (q, t) to the set I_{j_l},

We define the rank comparison relation $\triangleright \subseteq Q \times \mathbb{N}_0^{2k+1} \times \mathbb{N}_0^{2k+1}$ as follows: $(b,\ j_1{,}d_1, \ldots, j_k{,}d_k) \triangleright_q (b',\ j'_1{,}d'_1, \ldots, j'_k{,}d'_k)$ iff there exists $l \in [k]$ such that one of the following holds:

$$\begin{aligned} & b > b', \\ (b, \ldots, j_{l-1}, d_{l-1}) = (b', \ldots, j'_{l-1}, d'_{l-1}) \ & \wedge \ j_l > j'_l \ \wedge \ q \notin \bigcup_{m \in [l-1]} F_{j_m}, \\ (b, \ldots, j_l) = (b', \ldots, j'_l) \ & \wedge \ d_l > d'_l \ \wedge \ q \notin \bigcup_{m \in [l]} F_{j_m}, \\ (b, \ldots, j_l) = (b', \ldots, j'_l) \ & \wedge \ q \in I_{j_l} \ \wedge \ q \notin \bigcup_{m \in [l]} F_{j_m}. \end{aligned} \tag{3.4}$$

Here is the intutition. The first line bounds the number of visits to F_{j_1} (b decreases each time F_{j_1} is visited). The second line limits the changes of order $j_1, \ldots, j_k$ in the rank $(b,\ j_1, d_1, \ldots, j_k, d_k)$ to a finite number. Together, these two lines ensure that on any path some F_m is not visited infinitely often. The third and fourth lines require I_{j_l} to be visited within d_l steps; once it is visited, the distance d_l can be reset to any number $\leq |Q \times T|$.

We can encode the rank comparison constraints in Eq. 3.4 into SMT as follows. For each of $j_1, \ldots, j_k$ introduce an uninterpreted function: $Q \times T \to [k]$. For each of $b, d_1, \ldots, d_k$ introduce an uninterpreted function: $Q \times T \to [0, |Q \times T|]$. Finally, replace in Eq. 3.4 counters b, j, d, b', j', d' with expressions $b(q,t)$, $j(q,t)$, $d(q,t)$, $b(q',t')$, $j(q',t')$, $d(q',t')$ resp.

Ranking for Generalized Automata

The extension to generalized automata is simple: replace $\rho(q,t) \triangleright \rho(q',t')$ with $\bigwedge_i \rho^i(q,t) \triangleright^i \rho^i(q',t')$ where ρ^i and $\triangleright^i$ are for ith automaton acceptance component.

3.2.6 Discussion of Ranking

A close work on rankings is the work by Beyene et al. [11] on solving *infinite*-state games using SMT solvers. Conceptually, they use co-Büchi and Büchi ranking functions to encode game winning into SMT, which was also partially done by Schewe and Finkbeiner [46] a few years earlier (for finite-state systems). The authors focused on co-Büchi and Büchi automata, while we also considered Rabin and Streett automata (for finite-state systems). Although they claimed their approach can be extended to μ-calculus (and thus to CTL*), they did not elaborate beyond noting that CTL* verification can be reduced to games. In the next section we introduce two approaches to bounded synthesis from CTL*. Both approaches inherit the ideas on rankings presented in this section.

3.3 Bounded Synthesis from CTL*

We describe two ways to encode model checking for CTL* into SMT. The first one, direct encoding (Section 3.3.1), resembles bottom-up CTL* model checking [32] (see also page 2). The second encoding (Section 3.3.2) follows the automata-theoretic approach [63] (see also Section 2.6) and goes via hesitant tree automata. As usual, replacing a concrete system function with an uninterpreted one of a fixed size gives a bounded synthesis procedure.

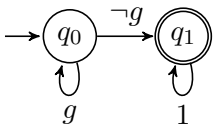
Let us compare the approaches. In the direct encoding, the main difficulty is the procedure that generates the constraints: we need to walk through the formula and generate constraints for nondeterministic Büchi or universal co-Büchi sub-automata. In the approach via hesitant tree automata, we first translate a given CTL* formula into a hesitant tree automaton A, and then encode the non-emptiness problem of the product of A and the system into an SMT query. In contrast to the direct encoding, the difficult—from the implementation point of view—part is to construct the automaton A, while the definition of the rank comparison relation is very easy.

In the next section we define CTL* with inputs and then describe two approaches. The approaches are conceptually the same, thus automata fans are invited to read Section 3.3.2 about the approach using hesitant automata, while the readers preferring bottom-up CTL* model checking are welcomed to Section 3.3.1.

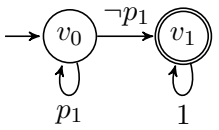
3.3.1 Direct Encoding

We reduce the CTL* model checking problem into SMT following the classical bottom-up model checking approach (see page 22).

Let $M = (I, O, T, t_0, \tau, out)$ be a machine and Φ be a CTL* state formula (in positive normal form). We use the notions of F, P, and $\widetilde{\Phi}$ defined on page 22: recall that with every state subformula $\mathsf{E}\varphi$ or $\mathsf{A}\varphi$ we associate a Boolean proposition, whose truth in a system state t implies that the corresponding subformula holds. The set $P = \{p_1, ..., p_k\}$ is the set of such propositions, the set $F = \{f_1, ..., f_k\}$ is the set of subformulas corresponding to $\{p_1, ..., p_k\}$ (note that each f_i is of the



(a) NBW for $\mathsf{F}\,\neg g$, associated with p_1 that encodes the truth of $\mathsf{EF}\,\neg g$.



(b) UCW for Gp_1, associated with p_2 that encodes the truth of $\mathsf{AG}\,p_1$.

Figure 3.1: Automata for Example 1

form $\mathsf{A}\varphi$ or $\mathsf{E}\varphi$ and φ has no path quantifiers), and $\widetilde{\Phi}$ is the top-level Boolean formula. We define the SMT query as follows.

(1) The query talks about uninterpreted functions $rch : Q_{all} \times T \to \mathbb{B}$, $\rho : Q_{all} \times T \to \mathbb{N}$, $\tau : T \times 2^I \to T$, $out : T \to 2^O$, and $p : T \to 2^P$. What is Q_{all} will become clear later.

(2) For each $f \in \{f_1, \ldots, f_k\}$, we do the following. If f is of the form $\mathsf{A}\varphi$, we translate φ into a UCW[2], otherwise into an NBW; let the resulting automaton be $A_\varphi = (2^{I \cup O \cup P}, Q, q_0, \delta, F)$. Note that $\delta \subseteq Q \times 2^I \times 2^O \times 2^P \times Q$, and it depends on P. For every $(q, t) \in Q \times T$, the query contains the constraints:

(2a) If A_φ is an NBW, then:

$$rch(q,t) \;\to \bigvee_{(e,q') \in \delta(q, out(t), p(t))} rch(q',t') \wedge \rho(q,t) \rhd_B \rho(q',t')$$

(2b) If A_φ is a UCW, then:

$$rch(q,t) \;\to \bigwedge_{(e,q') \in \delta(q, out(t), p(t))} rch(q',t') \wedge \rho(q,t) \rhd_C \rho(q',t')$$

In both cases, we have: $p(t) = \{p_i \in P \mid rch(q_0^{p_i}, t) = \mathsf{true}\}$, $q_0^{p_i}$ is the initial state of A_{φ_i}, $\rhd_B$ and $\rhd_C$ are the Büchi and co-Büchi rank comparison relations wrt. A_φ (see Eq. 3.1–3.2), and $t' = \tau(t, i)$. Intuitively $p(t)$ underapproximates the subformulas that hold in t: if $p_i \in p(t)$, then $t \models f_i$.

(3) The query contains the constraint $\widetilde{\Phi}[p_i \mapsto rch(q_0^{p_i}, t_0)]$, where $q_0^{p_i}$ is the initial state of A_{φ_i}. For example, for $\Phi = g \wedge \mathsf{AGEF}\,\neg g$ where $g \in O$, the constraint is $g(t_0) \wedge rch(q_0^{p_2}, t_0)$, where p_2 corresponds to $\mathsf{AG}\,p_1$, p_1 corresponds to $\mathsf{EF}\,\neg g$.

Example 1. Let $I = \{r\}$, $O = \{g\}$, $\Phi = g \wedge \mathsf{AGEF}\,\neg g$. We associate p_1 with $\mathsf{EF}\,\neg g$ and p_2 with $\mathsf{AG}\,p_1$. Automata for p_1 and p_2 are in Fig. 3.1, the SMT constraints are in Fig. 3.2.

Theorem 9 (Correctness of direct encoding)**.** *Given a CTL* formula Φ over inputs I and outputs O and a system $M = (I, O, T, t_0, \tau, out)$: $M \models \Phi$ iff the SMT query is satisfiable.*

[2]To translate φ into a UCW, translate $\neg\varphi$ into an NBW and treat it as a UCW.

$$
\begin{aligned}
\text{initial}: &\quad g(t_0) \wedge rch(q_0, t_0) \\
v_0 \xrightarrow{p_1} v_0: &\quad rch(v_0,t) \wedge rch(q_0,t) \rightarrow \bigwedge_{r \in \mathbb{B}} rch(v_0, \tau(t,r)) \wedge \rho(v_0,t) \geq \rho(v_0, \tau(t,r)) \\
v_0 \xrightarrow{\neg p_1} v_1: &\quad rch(v_0,t) \wedge \neg rch(q_0,t) \rightarrow \bigwedge_{r \in \mathbb{B}} rch(v_1, \tau(t,r)) \wedge \rho(v_0,t) \geq \rho(v_1, \tau(t,r)) \\
v_1 \xrightarrow{\mathsf{true}} v_1: &\quad rch(v_1,t) \rightarrow \bigwedge_{r \in \mathbb{B}} rch(v_1, \tau(t,r)) \wedge \rho(v_1,t) > \rho(v_1, \tau(t,r)) \\
q_0 \xrightarrow{g} q_0: &\quad rch(q_0,t) \wedge g(t) \rightarrow \bigvee_{r \in \mathbb{B}} rch(q_0, \tau(t,r)) \wedge \rho(q_0,t) > \rho(q_0, \tau(t,r)) \\
q_0 \xrightarrow{\neg g} q_1: &\quad rch(q_0,t) \wedge \neg g(t) \rightarrow \bigvee_{r \in \mathbb{B}} rch(q_1, \tau(t,r)) \wedge \rho(q_0,t) > \rho(q_1, \tau(t,r)) \\
q_1 \xrightarrow{\mathsf{true}} q_1: &\quad rch(q_1,t) \rightarrow \bigvee_{r \in \mathbb{B}} rch(q_1, \tau(t,r))
\end{aligned}
$$

Figure 3.2: SMT constraints for Example 1 for some $t \in T$. The final query is the conjunction of the constraints for every $t \in T$. The first line is the initialisation, item (3). The second line encodes the transition $v_0 \xrightarrow{p_1} v_0$ of the automaton in Fig. 3.1b and corresponds to item (2b): since we do not know whether p_1 holds in state t, we add the assumption $rch(q_0, t)$.

Here is the intuition behind the proof. The standard bottom-up model checker (see page 2) marks every system state with state subformulas it satisfies. The model checker returns "Yes" iff the initial state satisfies the top-level Boolean formula. The direct encoding conceptually follows that approach. If for some system state t, $rch(q_0^{p_i}, t)$ holds, then t satisfies the state formula f_i corresponding to p_i. Thus, if the top-level Boolean constraint (3) holds, then $t_0 \models \Phi$. And vice versa: if a model checker returns "Yes", then the marking it produced can be used to satisfy the SMT constraints. Finally, the positive normal form of Φ allows us to get away with encoding of positive obligations only ($rch(q_0^{p_i}, t) \Rightarrow t \models f_i$), eliminating the need to encode $\neg rch(q_0^{p_i}, t) \Rightarrow t \models \neg f_i$.

3.3.2 Encoding via Alternating Hesitant Tree Automata

Let us recall how we can model check and synthesize systems from CTL* formulas (see also Section 2.6). First, we convert a given CTL* formula into an alternating hesitant tree automaton. Then we build the product between the system and the automaton—such a product is a 1-letter alternating hesitant word automaton. Then we check the non-emptiness of the product automaton. We show how to encode the latter check into an SMT query. Such an SMT query is satisfiable iff the product is non-empty (thus the system satisfies the formula). As before, if we want to do synthesis, we replace a given system with an unknown system of a fixed size. Then an SMT solver returns a model (from which we extract a system), if such exists, together with a proof of the non-emptiness.

It is worth refreshing the following definitions: AHT and AHW (Chapter 2,

pages 17 and 19), 1-AHW and model checking wrt. CTL* (Section 2.6).

Encoding non-emptiness of the product into SMT

We start by converting a given CTL* formula Φ into an AHT. Then we build the product between a given system and the AHT. Such a product is a 1-AHW. We are going to encode the non-emptiness of the 1-AHW into an SMT query.

Let us explain the idea of the encoding. Recall that the states of the 1-AHW can be partitioned into "existential" sets $Q^N_1, ..., Q^N_{k_N}$ and "universal" sets $Q^U_1, ..., Q^U_{k_U}$. Such sets are ordered and the 1-AHW transition function ensures the following: Every path in every run-tree of the 1-AHW gets trapped in some Q^N_i or in Q^U_j. Such a path π is accepting iff $\mathit{Inf}(\pi) \cap \mathit{Acc} \neq \emptyset$ for the case of Q^N_i (Büchi acceptance) or $\mathit{Inf}(\pi) \cap \mathit{Acc} = \emptyset$ for the case of Q^U_j (co-Büchi acceptance). We will build an SMT query where the SMT solver has to: (a) resolve nondeterminism in the 1-AHW, (b) ensure that every path in the resulting universal word automaton is accepting.

Consider a system $M = (I, O, T, t_0, \tau, \mathit{out})$ and an AHT $A = (2^O, 2^I, Q, q_0, \delta : Q \times 2^O \to \mathcal{B}^+(2^I \times Q), \mathit{Acc} \subseteq Q)$ that corresponds to a given CTL* formula Φ. We encode the non-emptiness of the product $M \otimes A$, which has the states $Q \times T$, into the following SMT query:

$$\begin{aligned} & \mathit{rch}(q_0, t_0) \wedge \\ \bigwedge_{(q,t) \in Q \times T} & \mathit{rch}(q,t) \to \delta\big(q, \mathit{out}(t)\big) \Big[(d, q') \mapsto \mathit{rch}(q', \tau(t,d)) \wedge \\ & \qquad\qquad \rho(q,t) \ \triangleright_{q,q'} \ \rho(q', \tau(t,d))\Big] \end{aligned} \tag{3.5}$$

where $\triangleright_{q,q'}$[3] is:

- if q and q' are in the same Q^N_i, then the Büchi rank comparison $\triangleright_B^{Q^N_i}$;
- if q and q' are in the same Q^U_i, then the co-Büchi rank comparison $\triangleright_C^{Q^N_i}$;
- otherwise, true.

Theorem 10. *Given a system $(I, O, T, t_0, \tau, \mathit{out})$ and CTL* formula Φ over inputs I and outputs O: system $\models \Phi$ iff the SMT query in Eq. 3.5 is satisfiable.*

Proof idea. Direction $\Rightarrow$. Let $(Q, q_0, \delta, \mathit{Acc})$ be the 1-AHW representing the product system$\otimes$AHT. We will use the following observation.

> Observation: *The 1-AHW non-emptiness can be reduced to solving the following 1-Rabin game.* The game states are Q, the game graph corresponds to δ, there is one Rabin pair (F, I) with $F = \mathit{Acc} \cap Q^U$, $I = (\mathit{Acc} \cap Q^N) \cup (Q^U \setminus \mathit{Acc})$. Let us view δ to be in the DNF. Then, in state q of the game, the "existential" player (Automaton) chooses

[3]Here $\triangleright_{q,q'}$ depends on q *and* q', but it can also be defined to depend on q only, as it is originally introduced.

a disjunct in $\delta(q)$, while the "universal" player (Pathfinder) chooses a state in that disjunct. Automaton's strategy is winning iff for any Pathfinder's strategy the resulting play satisfies the Rabin acceptance (F, I). Note that Automaton has a winning strategy iff the 1-AHW is non-empty; also, memoryless strategies suffice for Automaton.

Since the 1-AHW is non-empty, Automaton has a memoryless winning strategy. We will construct *rch* and ρ from this strategy. For *rch*: set it to true if there is a strategy for Pathfinder such that the state will reached. Let us prove that ρ exists.

Since states from different Q_i can never form a cycle (due to the partial order), ρ of states from different Q_i are independent. Hence we consider two cases separately: ρ for some Q_i^N and for some Q_i^U.

- The case of Q_i^N is simple: by the definition of the 1-AHW, we can have only simple loops within Q_i^N. Any such reachable loop visits some state from $Acc \cap Q_i^N$. Consider such a loop: assign ρ for state q of the loop to be the minimal distance from any state $Acc \cap Q_i^N$.

- The case of Q_i^U: in contrast, we can have simple and non-simple loops within Q_i^U. But none of such loops visits $Acc \cap Q_i^U$. Then, for each $q \in Q_i^U$ assign ρ to be the maximum bad-distance from any state of Q_i^U. The bad-distance between q and q' is the maximum number of $Acc \cap Q^U$ states visited on any path from q to q'.

Direction $\Leftarrow$. The query is satisfiable means there is a model for *rch*. Note that the query is Horn-like ($\ldots \rightarrow \ldots$), hence there is a minimal marking *rch* of states that still satisfies the query[4][5]. Wlog., assume *rch* is minimal. Consider the subset of the states of the 1-AHW that are marked with *rch*, and call it U. Note that U is a 1-AHW and it has only universal transitions (i.e., we never mark more than one disjunct of δ on the right side of $\ldots \rightarrow \delta(\ldots)$). Intuitively, U is a finite-state representation of the (infinite) run-tree of the original 1-AHW.

Claim: *the run-tree—the unfolding of U—is accepting.* Suppose it is not: there is a run-tree path that violates the acceptance. Consider the case when the path is trapped in some Q_i^U. Then the path visits a state in $Q_i^U \cap Acc$ infinitely often. But this is impossible since we use co-Büchi ranking for Q_i^U. Contradiction. The case when the path is trapped in some Q_i^N is similar—the Büchi ranking prevents from not visiting $Acc \cap Q_i^N$ infinitely often.

Thus, the 1-AHW is non-empty since it has an accepting run-tree (the unfolding of U). □

[4]Minimal in the sense that it is not possible to reduce the number of *rch* truth values by falsifying some of *rch*.

[5]Non-minimality appears when δ of the alternating automaton has OR and the SMT solver marks with *rch* more than one OR argument. Another case is when the solver marks some state with *rch* but there is no antecedent requiring that.

3.3.3 Prototype Synthesizer for CTL*

We implemented both approaches to CTL* synthesis described in Sections 3.3.1 and 3.3.2 inside the tool PARTY [57]: https://github.com/5nizza/party-elli (branch "cav17"). In this section we illustrate the approach via AHTs.

The synthesizer works as follows:

(1) Parse the specification that describes inputs, outputs, and a CTL* formula Φ.

(2) Convert Φ into a hesitant tree automaton using the procedure described in [63], using LTL3BA [8] to convert path formulas into NBWs.

(3) For each system size k in increasing order (up to some bound):

- encode "$\exists M_k : M_k \otimes AHT \neq \emptyset$?" into SMT using Eq. 3.5 where $|M_k| = k$
- call Z3 solver [34]: if the solver returns "unsatisfiable", goto next iteration; otherwise print the system in the dot graph format.

This procedure is complete, because there is a $O(2^{2^{|\Phi|}})$ bound on the size of the system, although reaching it is impractical.

Running example: resettable 1-arbiter. Let $I = \{r\}$, $O = \{g\}$. Consider a simple CTL* property of an arbiter

$$\mathsf{EG}(\neg g) \wedge \mathsf{AG}(r \rightarrow \mathsf{F}\, g) \wedge \mathsf{AG}\,\mathsf{EF}\,\neg g.$$

The property says: there is a path from the initial state where the system never grants (including the initial state); every request should be granted; and finally, a state without the grant should always be reachable. We now invite the reader to Figure 3.3. It contains the AHT produced by our tool, and on its right side we show the product of the AHT with the one-state system that does not satisfy the property. The correct system needs at least two states and is on Figure 3.4.

Resettable 2-arbiter. Let $I = \{r_1, r_2\}$, $O = \{g_1, g_2\}$. Consider the formula

$$\begin{aligned}&\mathsf{EG}(\neg g_1 \wedge \neg g_2) \ \wedge \mathsf{AG}\,\mathsf{EF}(\neg g_1 \wedge \neg g_2) \wedge \\ &\mathsf{AG}(r_1 \rightarrow \mathsf{F}\, g_1) \wedge \mathsf{AG}(r_2 \rightarrow \mathsf{F}\, g_2) \wedge \mathsf{AG}(\neg(g_1 \wedge g_2)).\end{aligned}$$

Note that without the properties with E, the synthesizer can produce the system in Figure 3.5a which starts in the state without grants and then always grants one or another client. Our synthesizer outputs the system in Figure 3.5b (in one second).

Sender-receiver system. Consider a sender-receiver system of the following structure. It has two modules, the sender (S) with inputs $\{i_1, i_2\}$ and output *wire* and the receiver (R) with input *wire* and outputs $\{o_1, o_2\}$.

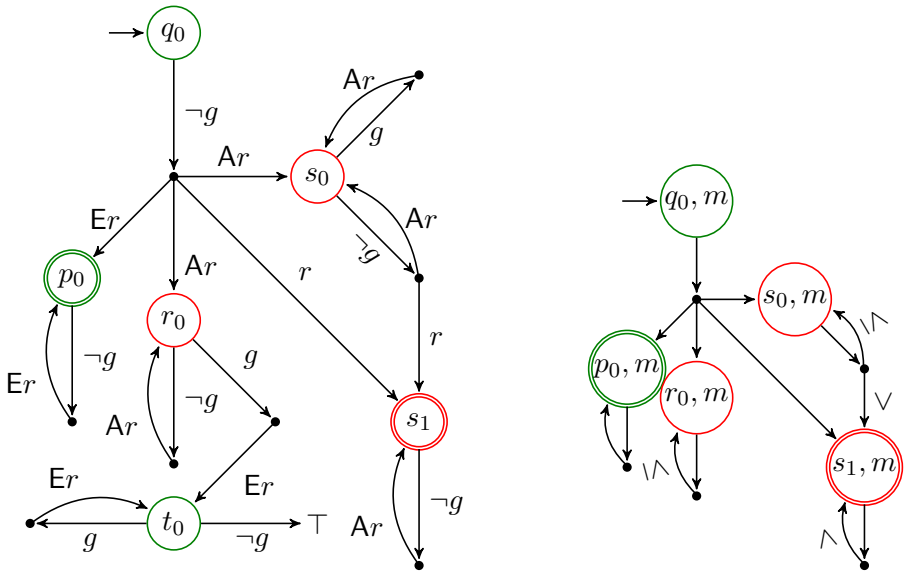
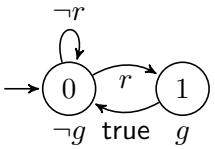
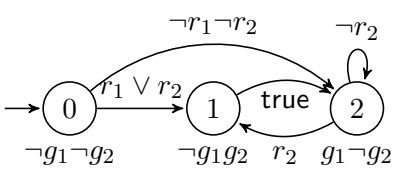


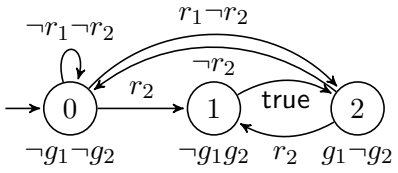
Figure 3.3: On the left: AHT for the CTL* formula $\mathsf{EG}\,\neg g \wedge \mathsf{AG}\,\mathsf{EF}\,\neg g \wedge \mathsf{AG}(r \rightarrow \mathsf{F}\,g)$. Green states are from the nondeterministic partion, red states are from the universal partition, double states are final (a red final state is rejecting, a green final state is accepting). State $\top$ denotes an accepting state. All transitions going out of the black dots are conjuncted. For example, $\delta(q_0, \neg g) = ((r, p_0) \vee (\neg r, p_0)) \wedge ((r, r_0) \wedge (\neg r, r_0)) \wedge (r, s_1) \wedge ((r, s_0) \wedge (\neg r, s_0))$. States s_0 and s_1 describe the property $\mathsf{AG}(r \rightarrow \mathsf{F}\,g)$, state p_0 — $\mathsf{EG}\,\neg g$, states r_0 and t_0 — $\mathsf{AG}\,\mathsf{EF}\,\neg g$, state t_0 — $\mathsf{EF}\,\neg g$. (Note that some states, e.g. q_0, do not have a transition for some letters. We assume that non-existing transitions go into a non-rejecting self-loop state for red (universal) states, and into a non-accepting self-loop state for green (nondeterministic) states.)
On the right side is the product (1-AHW) of the AHT with the one state system that never grants (thus it has $m \xrightarrow{\mathsf{true}} m$ and $out(m) = \neg g$). The edges are labeled with the relation $\triangleright_{q,q'}$ defined in Eq. 3.5. The product has no plausible annotation due to the cycle $(s_1, m) \xrightarrow{\geq} (s_1, m)$, thus the system does not satisfy the property.

Figure 3.4: The system that satisfies the property of the resettable 1-arbiter.



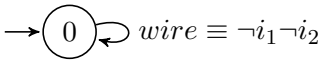


(a) The case without E properties

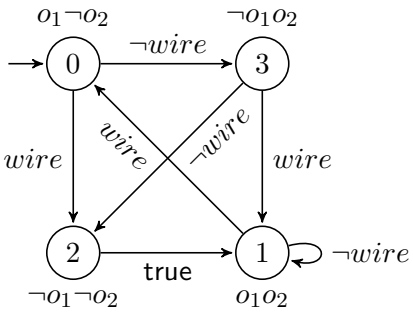


(b) The case with E properties

Figure 3.5: Synthesized systems for the resettable arbiter example



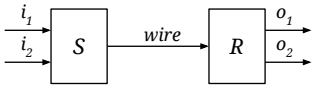
(a) The sender (Mealy machine)



(b) The receiver (Moore machine)

Figure 3.6: The synthesized system for the sender-receiver example

The sender can send one bit over the wire to the receiver. We would like to synthesize the sender and receiver modules that satisfy the following CTL* formula over $I = \{i_1, i_2\}$ and $O = \{o_1, o_2\}$:



$$\mathsf{AG}((i_1 \wedge i_2) \rightarrow \mathsf{F}(o_1 \wedge o_2)) \wedge$$
$$\mathsf{AG}((i_1 \wedge i_2 \wedge o_1 \wedge o_2) \rightarrow \mathsf{X}(o_1 \wedge o_2)) \wedge$$
$$\mathsf{AG}(\ \mathsf{EF}(o_1 \wedge \neg o_2) \wedge \mathsf{EF}(\neg o_1 \wedge o_2) \wedge \mathsf{EF}(\neg o_1 \wedge \neg o_2) \wedge \mathsf{EF}(o_1 \wedge o_2)\ ).$$

Our tool does not support distributed synthesis, so we manually adapted the SMT query it produced, by introducing the following uninterpreted functions.

- For the sender: the transition function $\tau_s : T_s \times 2^{\{i_1, i_2\}} \rightarrow T_s$ and the output function $out_s : T_s \times 2^{\{i_1, i_2\}} \rightarrow \mathbb{B}$. We set T_s to have a single state.
- For the receiver: the transition function $\tau_r : T_r \times 2^{\{wire\}} \rightarrow T_r$ and the output functions $o_1 : T_r \rightarrow \mathbb{B}$ and $o_2 : T_r \rightarrow \mathbb{B}$. We set T_r to have four states.

It took Z3 solver about 1 minute to find the solution shown in Figure 3.6.

3.3.4 Discussion of Bounded Synthesis from CTL*

We described two approaches to the CTL* synthesis and the only (to our knowledge) synthesizer supporting CTL*. (For CTL synthesis see [60, 33, 74], and [10] for PCTL.) The two approaches are conceptually similar. The approach via direct encoding is easier to code. The approach via alternating hesitant automata

hints, for example, at how to reduce CTL* synthesis to solving safety games: via bounding the number of visits to co-Büchi final states and bounding the distance to Büchi final states, and then determinizing the resulting automaton. A possible future direction is to extend the approach to the logic ATL* and distributed systems. In the next chapter, we show how CTL* synthesis can be reduced to LTL synthesis, which avoids developing specialized CTL* synthesisers, presented here.

Chapter 4

CTL* Synthesis via LTL Synthesis

This chapter is based on joint work with Roderick Bloem and Sven Schewe [23].

Abstract. We reduce synthesis for CTL* properties to synthesis for LTL. In the context of model checking this is impossible — CTL* is more expressive than LTL. Yet, in synthesis we have knowledge of the system structure *and* we can add new outputs. These outputs can be used to encode witnesses of the satisfaction of CTL* subformulas directly into the system. This way, we construct an LTL formula, over old and new outputs and original inputs, which is realisable if, and only if, the original CTL* formula is realisable. The CTL*-via-LTL synthesis approach preserves the problem complexity, although it might increase the minimal system size. We implemented the reduction, and evaluated the CTL*-via-LTL synthesiser on several examples.

4.1 Introduction

The problem of reactive synthesis was introduced by Church for Monadic Second Order Logic [28]. Later Pnueli introduced Linear Temporal Logic (LTL) [70] and together with Rosner proved 2EXPTIME-completeness of the reactive synthesis problem for LTL [71]. In parallel, Emerson and Clarke introduced Computation Tree Logic (CTL) [31], and later Emerson and Halpern introduce Computation Tree Star Logic (CTL*) [41] that subsumes both CTL and LTL. Kupferman and Vardi showed [62] that the synthesis problem for CTL* is 2EXPTIME-complete.

Intuitively, LTL allows one to reason about infinite computations. The logic has *temporal* operators, e.g., G (always) and F (eventually), and allows one to state properties like "every request is eventually granted" ($\mathsf{G}(r \rightarrow \mathsf{F}\, g)$). A system satisfies a given LTL property if *all* its computations satisfy it.

In contrast, CTL and CTL* reason about computation trees, usually derived by unfolding the system. The logics have—in addition to temporal operators—*path quantifiers*: A (on all paths) and E (there exists a path). CTL forbids arbitrary nesting of path quantifiers and temporal operators: they must interleave. E.g., $\mathsf{AG}\, g$ ("on all paths we always grant") is a CTL formula, but $\mathsf{AGF}\, g$ ("on all paths we infinitely often grant") is not a CTL formula. CTL* lifts this limitation.

The expressive powers of CTL and LTL are incomparable: there are systems indistinguishable by CTL but distinguishable by LTL, and vice versa. One important property inexpressible in LTL is the resettability property: "there is always a way to reach the 'reset' state" ($\mathsf{AGEF}\, reset$).

There was a time when CTL and LTL competed for "best logic for model checking" [81]. Nowadays most model checkers use LTL, because it is easier for designers to think about paths rather than about trees. LTL is also prevalent in reactive synthesis. SYNTCOMP [1]—the reactive synthesis competition with the goal to popularise reactive synthesis—has two distinct tracks, and both use LTL (or variants) as their specification language.

Yet LTL leaves the designer without *structural* properties. One solution is to develop general CTL* synthesisers like the one we developed in Chapter 3. Another solution is to transform the CTL* synthesis problem into the form understandable to LTL synthesisers, i.e., to reduce CTL* synthesis to LTL synthesis. Such a reduction would automatically transfer performance advances in LTL synthesisers to a CTL* synthesiser. In this chapter we show one such reduction.

Our reduction of CTL* synthesis to LTL synthesis works as follows.

First, recall how the standard CTL* model checking works (see page 2). The verifier introduces a proposition for every state subformula—formulas starting with an A or an E path quantifier—of a given CTL* formula. Then the verifier annotates system states with these propositions, in the bottom up fashion, starting with propositions that describe subformulas over original propositions (system inputs and outputs). Therefore the system satisfies the CTL* formula iff the initial system state is annotated with the proposition describing the whole CTL* formula (assuming that the CTL* formula starts with A or E).

Now let us look into CTL* synthesis. The synthesiser has the flexibility to choose the system structure, as long as it satisfies a given specification. We introduce new propositions—outputs that later can be hidden from the user—for state subformulas of the CTL* formula, just like in the model checking case above. We also introduce additional propositions for existentially quantified subformulas—to encode the witnesses of their satisfaction. Such propositions describe the directions (inputs) the environment should provide to satisfy existentially quantified path formulas. The requirement that new propositions indeed denote the truth of the subformulas can be stated in LTL. For example, for a state subformula $\mathsf{A}\varphi$, we introduce proposition $p_{\mathsf{A}\varphi}$, and require $\mathsf{G}\,[p_{\mathsf{A}\varphi} \rightarrow \varphi']$, where φ' is φ with state subformulas substituted by the propositions. For an existential subformula $\mathsf{E}\varphi$, we introduce proposition $p_{\mathsf{E}\varphi}$ and require, *roughly*, $\mathsf{G}\left[p_{\mathsf{E}\varphi} \rightarrow ((\mathsf{G}\, d_{p_{\mathsf{E}\varphi}}) \rightarrow \varphi')\right]$, which states: if the proposition $p_{\mathsf{E}\varphi}$ holds, then the path along directions encoded by $d_{p_{\mathsf{E}\varphi}}$ satisfies φ' (where φ' as before). We wrote "roughly", because there can be several different witnesses for the same existential subformula starting at different system states: they may meet in the same system state, but depart afterwards—then, to be able to depart from the meeting state, each witness should have its own direction d. We show that, for each existential subformula, a number $\approx 2^{|\Phi_{\mathrm{CTL}^*}|}$ of witnesses is sufficient, where Φ_{CTL^*} is a given CTL* formula. This makes the LTL formula exponential in the size of the CTL* formula, but the special—conjunctive—nature of the LTL formula ensures that the synthesis complexity is 2EXPTIME wrt.

$|\Phi_{\text{CTL}^*}|$.

Our reduction is "if and only if" and preserves the synthesis complexity. However, it may increase the size of the system, and is not very well suited to establish unrealisability. Of course, to show that a given CTL* formula is unrealisable, one could reduce CTL* synthesis to LTL synthesis, then reduce the LTL synthesis problem to solving parity games, and derive the unrealisability from there[1]. But the standard approach for unrealisability checking—by synthesising the dualised LTL specification—does not seem to be practical. The reason is that the LTL formula Φ_{LTL} is exponential in the size $|\Phi_{\text{CTL}^*}|$ of the CTL* formula. The negated LTL formula $\neg\Phi$ (used in the dualised specification) is a big disjunction (vs. big conjunction for Φ_{LTL}), which makes a corresponding universal co-Büchi automaton doubly-exponential in $|\Phi_{\text{CTL}^*}|$ (vs. singly-exponential for Φ_{LTL}). The double exponential blow up in the size of the automaton—which is used as input to bounded synthesis—makes this unrealisability check impractical [2].

Finally, we have implemented[3] the converter from CTL* into LTL, and evaluated our CTL*-via-LTL synthesis approach, using two LTL synthesisers and CTL* synthesiser (Chapter 3), on several examples. The experimental results show that such an approach works very well—outperforming the specialised CTL* synthesiser (Chapter 3)—when the number of CTL*-specific formulas is small.

The chapter depends on notions defined in Chapter 2 and is structured as follows. In the next Section 4.2 we present the main contribution: the reduction. Then Section 4.3 briefly discusses checking unrealisability of CTL* specifications. Section 4.4 describes the experimental setup, specifications, solvers used, and synthesis timings, and Section 4.5 concludes.

4.2 Converting CTL* to LTL for Synthesis

In this section, we describe how and why we can reduce CTL* synthesis to LTL synthesis. First, we recall the standard approach to CTL* synthesis, then describe, step by step, the reduction and the correctness argument, and then discuss some properties of the reduction.

4.2.1 LTL Encoding

Let us first look at standard automata based algorithms for CTL* synthesis [62]. When synthesising a system that realizes a CTL* specification, we normally do the following.

- We turn the CTL* formula into an alternating hesitant tree automaton A.
- Move from computation trees to annotated computation trees that move

[1] Reducing LTL synthesis to solving parity games *is* practical, as SYNTCOMP'17 [1] showed: such synthesiser `ltlsynt` was among the fastest.

[2] This is a conjecture: we have *not* proven that the synthesis of dualised LTL formulas, produced by our reduction, takes triply exponential time.

[3] Available at https://github.com/5nizza/party-elli, branch "cav17"

the (memoryless) strategy of the verifier[4] into the label of the computation tree. This allows for using the derived universal co-Büchi tree automaton U, making the verifier deterministic: it does not make any decisions, as they are now encoded into the system;

- Determinise U to a deterministic tree automaton D.
- Play an emptiness game for D.
- If the verifier wins, his winning strategy (after projection of the additional labels) defines a system, if the spoiler wins, the specification is unrealisable.

We draw from this construction and use particular properties of the alternating hesitant tree automaton A. Namely, A is not a general alternating tree automaton, but is an alternating hesitant tree automaton. Such an automaton is built from a mix of nondeterministic Büchi and universal co-Büchi word automata, called "existential word automata" and "universal word automata". These universal and existential word automata start at any system state [tree node] where a universally or existentially, respectively, quantified subformula is marked as true in the annotated system [annotated computation tree]. We use the term "existential word automata" to emphasise that the automaton is not only a non-deterministic word automaton, but it is also used in the alternating tree automaton in a way, where the verifier can pick the system [tree] path, along which it has to accept.

We will use the following notions defined on page 22: the set F of state subformulas of a given CTL* formula Φ, the set of corresponding propositions P, and the top-level Boolean formula $\widetilde{\Phi}$.

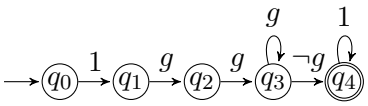
Example 2 (Word and tree automata). Consider the formula $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$ where inputs $I = \{r\}$ and outputs $O = \{g\}$. The set $F = \{f_{\mathsf{EG}} = \mathsf{EG}\,p_{\mathsf{EX}}, f_{\mathsf{EX}} = \mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))\}$, the set $P = \{p_{\mathsf{EX}}, p_{\mathsf{EG}}\}$, and $\widetilde{\Phi} = p_{\mathsf{EG}}$. Figure 4.1 shows the nondeterministic word automata for the path formulas of the subformulas, and the alternating (actually, nondeterministic) tree automaton for the whole formula. In what follows, we work mostly with word automata.

We are going to show how and why we can reduce CTL*-synthesis to LTL synthesis. The argument is split into steps (a), (b), (c), (d), and (e). Figure 4.2 summarises the steps.

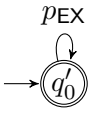
Step A (the starting point). The verifier takes as input: a computation tree, universal and existential word automata for the CTL* subformulas, and the top-level proposition corresponding to the whole CTL* formula. It has to produce an accepting run tree (if the computation tree satisfies the formula).

Step B. Given a computation tree, the verifier maps each tree node to a (universal or existential word) automaton state, and moves from a node according to the quantification of the automaton (either in all tree directions or in one direction). The decision in which tree direction to move and which automaton state to pick for the successor node, constitutes the strategy of the verifier. Each time the verifier

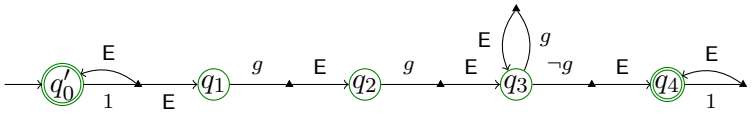
[4]Such a strategy maps, in each tree node, an automaton state to a next automaton state and direction.



(a) NBW for $\mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$, the alphabet $\Sigma = 2^{\{r,g\}}$. Transitions to the non-accepting state *sink* are not shown.



(b) NBW for $\mathsf{G}(p_{\mathsf{EX}})$, the alphabet $\Sigma = 2^{\{r,g,p_{\mathsf{EX}}\}}$. The transition to the non-accepting state *sink* is omitted.



(c) Alternating hesitant tree automaton for $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$ (actually it is nondeterministic). The green color of the states indicate that they are from the nondeterministic partition of the states (and thus double-circled states are from the Büchi acceptance condition). The edges starting in the filled triangle are connected with $\wedge$. Edge label E abbreviates the set of edges, for each tree direction, connected with $\vee$. Thus, the transition from q'_0 is $((q'_0, r) \vee (q'_0, \neg r)) \wedge ((q_1, r) \vee (q_1, \neg r))$. To get an alternating automaton for $\mathsf{AG}\,\mathsf{EX}(...)$, replace in the self-loop edge of q'_0 label E with A, and make the state non-rejecting (these also move the state into the universal partition of the states).

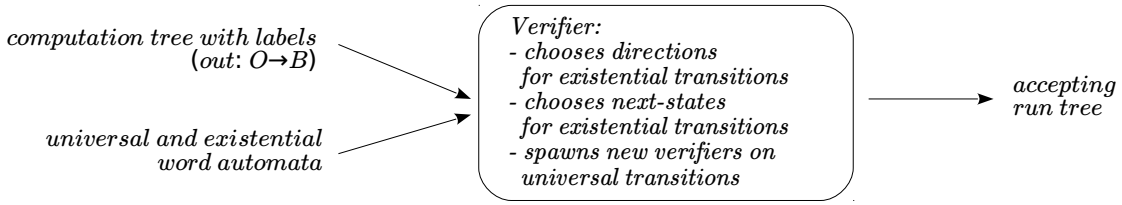
Figure 4.1: Word and tree automata.

has to move in several tree directions (this happens when the node is annotated with a *universal* word automaton state), we spawn a new version of the verifier, for each tree direction and transition of the universal word automaton.

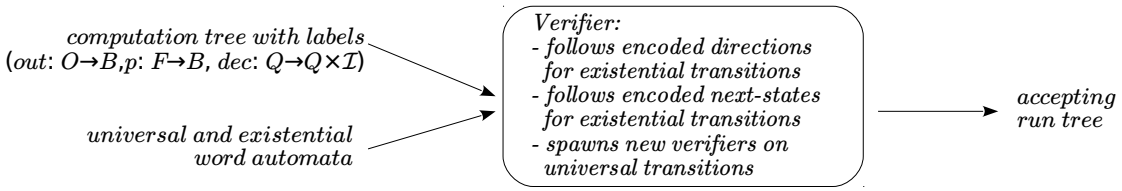
The strategy of the verifier is a mapping of states of the existential word automata to a decision, which consists of a tree direction (the continuation of the tree path, along which the automaton shall accept) and an automaton successor state transition. For every node n, this is a mapping $dec : Q \to 2^I \times Q$ such that $dec(q) = (e, q')$ implies that $q' \in \delta\big(q, (l(n), e)\big)$, where δ corresponds to the existential word automaton to which q belongs, and $l(n) \in 2^O$ is a label of the current tree node n[5]. Note that strategies are defined per-node-basis, i.e., *dec* may differ in two different nodes n_1 and n_2. (All node labels depend on the current node, but we will omit specifying this explicitly.) Notice that the strategy is memoryless wrt. the history of automata states.

We call a model, in which every state is additionally annotated with a verifier strategy, an *annotated model*. Similarly, an *annotated computation tree* is a computation tree in which every node is additionally annotated with a verifier strategy. Thus, in both cases, every system state [node] is labeled with: (i) original propositional labeling $out : O \to \mathbb{B}$, (ii) propositional labeling for universal and existential subformulas $F = F_{univ} \dot\cup F_{exist}$, $p : F \to \mathbb{B}$, and (iii) decision labeling $dec : Q \to 2^I \times Q$ where Q are the states of all existential automata.

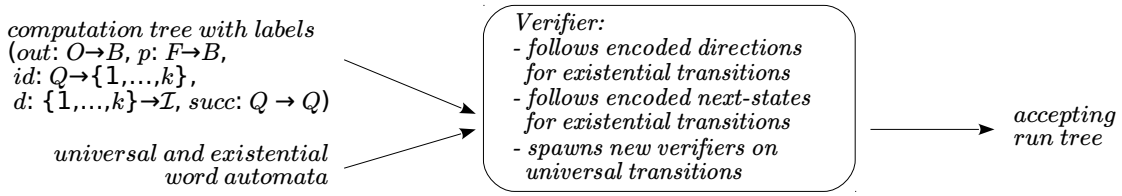
[5]The verifier, when in the tree node or system state, moves according to this strategy.



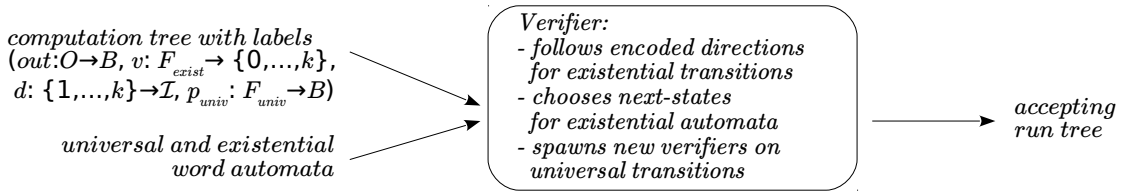
(a) The verifier takes a computation tree, universal and existential word automata, and the top-level proposition, that together encode a given CTL* formula. It produces an accepting run tree (if the computation tree satisfies the formula).



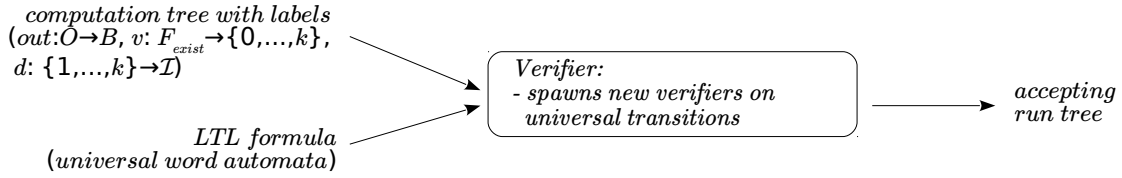
(b) We encode the verifier decisions into annotated computation trees, making the verifier deterministic. Figure 4.3b shows such an annotated computation tree.



(c) The new annotation is a re-phrasing of the previous one. Figure 4.4 gives an example.

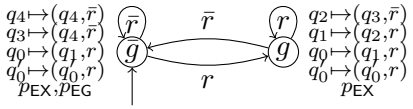


(d) We keep directions in the annotation but remove next-states—now the verifier has to choose. Figure 4.5 gives an example. The change from the label $id : Q \to \{1, ..., k\}$ to the label $v : F_{exist} \to \{0, ..., k\}$ is the reason why the system size can increase.

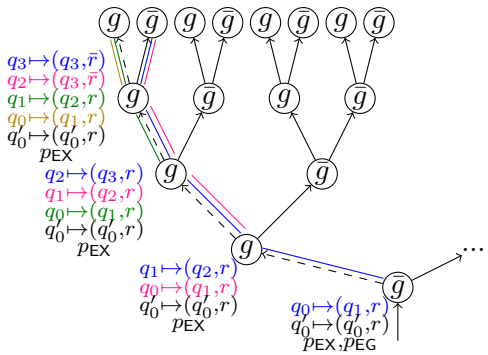


(e) Now the obligation of the verifier can be stated in LTL (or using universal co-Büchi word automata).

Figure 4.2: Steps in the correctness argument. We transform the input to the verifier and its task, step by step. We begin with a computation tree labeled with 2^O and end with a computation tree labeled additionally with $v : F_{exist} \to \{0, ..., k\}$ and $d : \{1, ..., k\} \to \mathcal{I}$. (Calligraphic $\mathcal{I}$ denotes 2^I.) This is about verifying a given computation tree (labels are fixed), in synthesis we would search for such a tree.



(a) An annotated system satisfying $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. Near the nodes is the annotation that encodes the winning strategy of the verifier, the label p_{EX} means the subformula $\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$ holds, the label p_{EG} means the subformula $\mathsf{EG}\,p_{\mathsf{EX}}$ holds.



(b) An annotated computation tree that satisfies $\mathsf{EG}\,\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. The root node is called ϵ, its left child r, the left child of node r is rr, and so on. Let p_{EG} correspond to $\mathsf{EG}(p_{\mathsf{EX}})$ and let p_{EX} correspond to $\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$. The annotation for the verifier strategy is on the left side of nodes, and decisions for non mapped states are irrelevant. Paths used by the winning strategy are depicted using dashed and colored lines. The black dashed path witnesses p_{EG}, the blue path witnesses p_{EX} starting in the root node ϵ, the pink path witnesses p_{EX} starting in the node r, and so on. The pink and blue paths share the tail. Note that this particular annotated computation tree is *not* the unfolding of the annotated system above: in the annotated system the right state maps $q_2 \mapsto (q_3, \bar{r})$, while in the tree the node rr has $q_2 \mapsto (q_3, r)$. (This is done to illustrate that mapped out tree paths can share the tail.)

Figure 4.3: Annotated system and computation tree.

Example 3. Figure 4.3 shows an annotated system and computation tree.

Step C. The verifier strategy (encoded in the annotated computation tree) encodes both the words on which the nondeterministic automata are interpreted and witnesses of acceptance (accepting automata paths on those words). For the encoding in LTL that we will later use, it is enough to map out the word, and replace the witness by what it actually means: that the automaton word satisfies the respective path formula. I.e., if a proposition p corresponding to an existential formula $\mathsf{E}\varphi$ holds in a tree node, then it will be enough to require that φ holds on the path starting in that node and that follows the directions encoded in the tree.

Example 4. Let us look at Figure 4.3b to understand the notions of mapped out paths and words. For every proposition marking a tree node there is a mapped out path. Consider the root node labeled with p_{EX} and p_{EG} and look at p_{EG} first. The proposition p_{EG} corresponds to $\mathsf{EG}\,p_{\mathsf{EX}}$ and is associated with the NBW in

Figure 4.1b that has the initial state q'_0. We consult the strategy $q'_0 \mapsto (q'_0, r)$ and move in direction r into node r (note that the root is labeled p_{EX} and thus we *can* transit $q'_0 \xrightarrow{r} q'_0$). In the node r we consult the strategy $q'_0 \mapsto (q'_0, r)$ and again move in direction r into node rr, and so on. This way we map out the tree path $\epsilon, \mathrm{r}, \mathrm{rr}, ...$ for p_{EG} from the root, and the corresponding mapped out word is $(\bar{g}, p_{\mathsf{EX}}, r)(g, p_{\mathsf{EX}}, r)^\omega$. Now consider the root label p_{EX} that corresponds to $\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$ and is associated with the NBW in Figure 4.1a. We consult the strategy $q_0 \mapsto (q_1, r)$ that tells us to move in direction r into node r (again, note that the root label g makes it possible to transit $q_0 \xrightarrow{g} q_1$). In the node r we consult the strategy $q_1 \mapsto (q_2, r)$ and move into node rr, while the automaton state advances to q_2. From the node rr the strategy $q_2 \mapsto (q_3, r)$ directs us into node rrr, then the node rrr has the strategy $q_3 \mapsto (q_3, \bar{r})$, and so on. Thus, from the root for the proposition p_{EX} the strategy maps out the path $\epsilon, \mathrm{r}, \mathrm{rr}, \mathrm{rrr}, \mathrm{rr}\bar{\mathrm{r}}, ...$ and the word $(\bar{g}, r)(g, r)(g, r)(g, \bar{r})(\bar{g}, \bar{r})^\omega$.

Let *two tree paths be equivalent* if they share a tail (equivalently, if one is the tail of the other). Our interest will be in equivalence of mapped out tree paths.

There is a simple sufficient condition for two mapped out tree paths to be equivalent: if they pass through the same node of the annotated computation tree in the same automaton state, then they have the same future, and are therefore equivalent. The condition is sufficient but not necessary[6].

Example 5. In Figure 4.3b the blue and pink paths are equivalent, since they share a tail. The sufficient condition fires in the node $\mathrm{rrr}\bar{\mathrm{r}}$, where the tree paths meet in the automaton state q_3

The sufficient condition implies that we cannot have more non-equivalent tree paths passing through a tree node than there are states in all existential word automata, let us call this number k: $k = sum_{\mathsf{E}\varphi \in F_{exist}} |Q_\varphi|$, where Q_φ are the states of an NBW for φ. For each tree node, we assign unique numbers from $\{1, ..., k\}$ to equivalence classes, and thus any two non-equivalent tree paths that go through the same tree node have different numbers. As this is an intermediate step in our translation, we are wasteful with the labeling:

(1) for every node n, we map existential word automata states to numbers (IDs) using $id : Q \to \{1, \ldots, k\}$, we also use labels $d : \{1, \ldots, k\} \to 2^I$ ("direction to take") and $succ : Q \to Q$ ("successor to take"), such that $succ(q) \in \delta\Big(q, \big(l(n), d(id(q))\big)\Big)$, *and*

(2) we maintain the same state ID along the chosen direction: $id_n(q) = id_{n \cdot e}(succ_n(q))$, where the subscript denotes a node to which the label belongs and $e = d_n(id_n(q))$.

Note that every annotated computation tree can be re-labeled in the above way. Indeed: the item (1) alone can be viewed as a re-phrasing of the labeling *dec* that

[6]Recall that each mapped out tree path corresponds to at least one copy of the verifier that ensures the path is accepting. When two verifiers go along the same tree path, it can be annotated with different automata states (for example, corresponding to different automata). Then such paths do not satisfy the sufficient condition, although they are trivially equivalent.

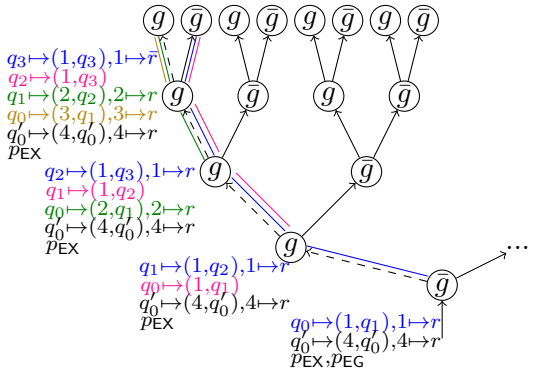


Figure 4.4: A re-labeled computation tree. Notation "$q_0 \mapsto (1, q_1)$" means $id(q_0) = 1$ and $succ(q_0) = q_1$, and "$1 \mapsto r$" means d maps 1 to $\{r\}$. Since the blue and pink paths are equivalent, the label id maps the corresponding automata states in the nodes to the same number, 1. The IDs of the green and yellow paths differ implying that they are not equivalent and hence do not share the tail (their tails cannot be seen in the figure).

we had before on page 49, and the requirement (2) is satisfiable because a tree path maintains its equivalence class. This step is shown in Figure 4.2c, the labels are: $(out : O \to \mathbb{B}, p : F \to \mathbb{B}, id : Q \to \{1, \ldots, k\}, d : \{1, \ldots, k\} \to 2^I, succ : Q \to Q)$.

Figure 4.4 shows a re-labeled computation tree of Figure 4.3b.

Step D. In the new annotation with labels $(out, p, id, d, succ)$, labeling d alone maps out the tree path for each ID. The remainder of the information is mainly there to establish that the corresponding word is accepted by the respective word automaton (equivalently: satisfies the respective path formula). If we use only d, then the only missing information is where the path starts and which path formula it belongs to—the information originally encoded by p.

We address these two points by using *numbered* computation trees. Recall that the annotated computation trees have a *propositional* labeling $p : F \to \mathbb{B}$ that labels nodes with subformulas. In the numbered computation trees, we replace p for *existential* subformulas $F_{exist} \subseteq F$ by labeling $v : F_{exist} \to \{0, ..., k\}$, where for every existentially quantified formula $\mathsf{E}\varphi \in F_{exist}$ and a tree node n:

- $v_{\mathsf{E}\varphi,n} = 0$ encodes that no claim that $\mathsf{E}\varphi$ holds in n is made (similarly to the proposition $p_{\mathsf{E}\varphi}$ being false in the annotated tree), whereas
- a value $v_{\mathsf{E}\varphi,n} \in \{1, ..., k\}$ requires that the word of a tree path with ID $v_{\mathsf{E}\varphi,n}$ starting in n and that follows $d(v_{\mathsf{E}\varphi,n})$ satisfies φ, i.e., the word corresponding to $n, n{\cdot}d_n(v_{\mathsf{E}\varphi,n}), n{\cdot}d_n(v_{\mathsf{E}\varphi,n}){\cdot}d_{nd_n(v_{\mathsf{E}\varphi,n})}(v_{\mathsf{E}\varphi,n}), ...$ satisfies φ (where d_n denotes d in node n).

Example 6. The tree in Figure 4.4 becomes a numbered computation tree if we replace the propositional labels p_{EX} and p_{EG} with ID numbers as follows. The root ϵ has $v_{\mathsf{EX}} = 1$ and $v_{\mathsf{EG}} = 4$, the left child r has $v_{\mathsf{EX}} = 1$, the node rr has $v_{\mathsf{EX}} = 2$, the node rrr has $v_{\mathsf{EX}} = 3$. Note that $id(q_0) = v_{\mathsf{EX}}$ and $id(q'_0) = v_{\mathsf{EG}}$ whenever those vs are non-zero. The nodes outside of the dashed path have $v_{\mathsf{EX}} = v_{\mathsf{EG}} = 0$, meaning that no claims about satisfaction of the corresponding path formulas is made.

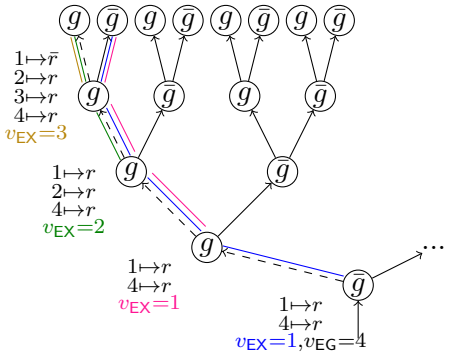


Figure 4.5: Numbered computation tree with redundant annotations removed.

Initially, we use *ID* labeling v in addition with $(out, id, d, succ, p^{univ})$, where p^{univ} is a restriction of p on F_{univ}, and then there is no relevant change in the way the (deterministic) verifier works. I.e., a numbered computation tree can be turned into annotated computation tree, and vice versa, such that the numbered tree is accepted iff the annotated tree is accepted.

Now we observe that the labelings *id* and *succ* are used only to witness that each word mapped out by d is accepted by respective existential word automata. I.e., *id* and *succ* make the verifier deterministic. Let us remove *id* and *succ* from the labeling. We call such trees *lean-numbered computation trees*; they have labeling $(out : O \to \mathbb{B}, v : F_{exist} \to \{0, ..., k\}, d : \{1, ..., k\} \to 2^I, p^{univ} : F_{univ} \to \mathbb{B})$. This makes the verifier nondeterministic. We still have the property that every accepting annotated computation tree can be turned into an accepting lean-numbered computation tree, and vice versa. This step is shown in Figure 4.2d; an example of a lean-numbered computation tree is in Figure 4.5.

Step E (the final step). We show how labeling (out, v, d, p^{univ}) allows for using LTL formulas instead of directly using automata for the acceptance check. The encoding into LTL is as follows.

- For each existentially quantified formula $\mathsf{E}\varphi$, we introduce the following LTL formula (recall that $v_{\mathsf{E}\varphi} = 0$ encodes that we do *not* claim that $\mathsf{E}\varphi$ holds in the current tree node, and $v_{\mathsf{E}\varphi} \neq 0$ means that $\mathsf{E}\varphi$ does hold and φ holds if we follow $v_{\mathsf{E}\varphi}$-numbered directions):

$$\bigwedge_{j \in \{1,\ldots,k\}} \mathsf{G}\left[v_{\mathsf{E}\varphi} = j \;\to\; \left(\mathsf{G}\, d_j \to \varphi'\right)\right], \tag{4.1}$$

  where φ' is obtained from φ by replacing the subformulas of the form $\mathsf{E}\psi$ by $v_{\mathsf{E}\psi} \neq 0$ and the subformulas of the form $\mathsf{A}\psi$ by $p_{\mathsf{A}\psi}$.

- For each subformula of the form $\mathsf{A}\varphi$, we simply take

$$\mathsf{G}\left[p_{\mathsf{A}\varphi} \;\to\; \varphi'\right], \tag{4.2}$$

  where φ' is obtained from φ as before.

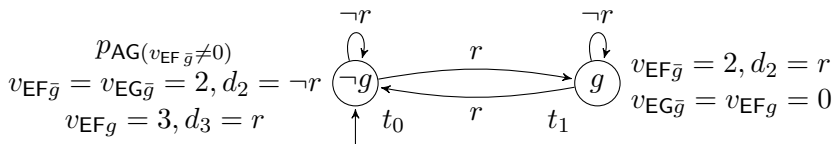


Figure 4.6: A Moore machine for Example 7. The witness for $\mathsf{EG}\,\neg g$ is: $v_{\mathsf{EG}\bar{g}}(t_0) = 2$, we move along $d_2 = \neg r$ looping in t_0, thus the witness is $(t_0)^\omega$. The witness for $\mathsf{EF}\,g$: since $v_{\mathsf{EF}g}(t_0) = 3$, we move along $d_3 = r$ from t_0 to t_1, where d_3 is not restricted, so let $d_3 = \neg r$ (not drawn) and then the witness is $t_0(t_1)^\omega$. The satisfaction of $\mathsf{AGEF}\,\neg g$ means that every state has $v_{\mathsf{EF}\bar{g}} \neq 0$, which is true. In t_0 we have $\neg g$, so $\mathsf{EF}\,\neg g$ is satisfied; for t_1 we have $v_{\mathsf{EF}\bar{g}}(t_1) = 2$ hence we move $t_1 \xrightarrow{r} t_0$ and $\mathsf{EF}\,\neg g$ is also satisfied.

- Finally, the overall LTL formula is the conjunction

$$\boxed{\Phi' \wedge \bigwedge_{\mathsf{E}\varphi \in F_{exist}} \text{Eq.4.1} \;\wedge \bigwedge_{\mathsf{A}\varphi \in F_{univ}} \text{Eq.4.2}} \tag{4.3}$$

  where the Boolean formula Φ' is obtained by replacing in the original CTL* formula every $\mathsf{E}\varphi$ by $v_{\mathsf{E}\varphi} \neq 0$ and every $\mathsf{A}\varphi$ by $p_{\mathsf{A}\varphi}$.

Example 7. Let $I = \{r\}$, $O = \{g\}$. Consider the CTL formula

$$\mathsf{EG}\,\neg g \wedge \mathsf{AG}\,\mathsf{EF}\,\neg g \wedge \mathsf{EF}\,g.$$

The sum of states of individual NBWs is 5 (assuming the natural translations), so we introduce integer propositions $v_{\mathsf{EF}\bar{g}}$, $v_{\mathsf{EG}\bar{g}}$, $v_{\mathsf{EF}g}$, ranging over $\{0, ..., 5\}$, and five Boolean propositions d_1, ..., d_5; we also introduce the Boolean proposition $p_{\mathsf{AG}(v_{\mathsf{EF}\bar{g}}\neq 0)}$. The LTL formula is:

$$\begin{aligned} & v_{\mathsf{EG}\bar{g}} \neq 0 \wedge p_{\mathsf{AG}(v_{\mathsf{EF}\bar{g}}\neq 0)} \wedge v_{\mathsf{EF}g} \neq 0 \;\wedge \\ & \bigwedge_{j\in\{1\ldots5\}} \mathsf{G}\begin{bmatrix} v_{\mathsf{EF}\bar{g}} = j \;\rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{F}\,\neg g) \\ v_{\mathsf{EG}\bar{g}} = j \;\rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{G}\,\neg g) \\ v_{\mathsf{EF}g} = j \;\rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{F}\,g) \end{bmatrix} \;\wedge \\ & \mathsf{G}\left[p_{\mathsf{AG}(v_{\mathsf{EF}\bar{g}}\neq 0)} \rightarrow \mathsf{G}(v_{\mathsf{EF}\bar{g}} \neq 0)\right]. \end{aligned}$$

Figure 4.6 shows a system satisfying the LTL specification.

Remark 7 (We *need* propositions for universal subformulas)**.** It is intuitively clear that we need new propositions for existential subformulas. But it is tempting to believe that we can skip introducing new propositions for universal subformulas and directly use the subformulas instead of the propositions. This is wrong. Consider the CTL* formula $\mathsf{AFAG}\,g$. Our reduction produces the LTL formula

$$p_{\mathsf{AF}} \wedge \mathsf{G}[p_{\mathsf{AF}} \rightarrow \mathsf{F}\,p_{\mathsf{AG}}] \wedge \mathsf{G}[p_{\mathsf{AG}} \rightarrow \mathsf{G}\,g].$$

If we substitute the new propositions with what they express (p_{AF} by $\mathsf{F}\,p_{\mathsf{AG}}$ and p_{AG} by $\mathsf{G}\,g$), then we get $\mathsf{FG}\,g$. But $\mathsf{AFAG}\,g$ is different from $\mathsf{AFG}\,g$.

The whole discussion leads us to the theorem.

Theorem 11. *Let I be the set of inputs and O be the set of outputs, and Φ_{LTL} be derived from a given Φ_{CTL^*} as described above. Then:*

$$\Phi_{CTL^*} \text{ is realisable} \quad \Leftrightarrow \quad \Phi_{LTL} \text{ is realisable.}$$

4.2.2 Complexity

The translated LTL formula Φ_{LTL}, due to Eq. 4.1, in the worst case, can be exponentially larger than Φ_{CTL^*}, $|\Phi_{\text{LTL}}| = 2^{\Theta(|\Phi_{\text{CTL}^*}|)}$. Yet, the upper bound on the size of $UCW_{\Phi_{\text{LTL}}}$ is $2^{\Theta(|\Phi_{\text{CTL}^*}|)}$ rather than $2^{\Theta(|\Phi_{\text{LTL}}|)} = 2^{2^{\Theta(|\Phi_{\text{CTL}^*}|)}}$, because:

- the size of the UCW is additive in the size of the UCWs of the individual conjuncts, and
- each conjunct UCW has almost the same size as a UCW of the corresponding subformula, since, for every LTL formula φ, $|UCW_{\mathsf{G}[p\rightarrow(\mathsf{G}d\rightarrow\varphi)]}| = |UCW_{\varphi}| + 1$.[7]

Determinising $UCW_{\Phi_{\text{LTL}}}$ gives a parity game with up to $2^{2^{\Theta(|\Phi_{\text{CTL}^*}|)}}$ states and $2^{\Theta(|\Phi_{\text{CTL}^*}|)}$ priorities [78, 68, 77]. The recent quasipolynomial algorithm [26] for solving parity games has a particular case for n states and $log(n)$ many priorities, where the time cost is polynomial in the number of game states. This gives us $O(2^{2^{|\Phi_{\text{CTL}^*}|}})$-time solution to the derived LTL synthesis problem. The lower bound comes from the 2EXPTIME-completeness of the CTL* synthesis problem [76].

Theorem 12. *Our solution to the CTL* synthesis problem via the reduction to LTL synthesis is 2EXPTIME-complete.*

Minimality

Although the reduction to LTL synthesis preserves the complexity class, it does not preserve the minimality of the systems. Consider an existentially quantified formula $\mathsf{E}\varphi$. A system path satisfying the formula may pass through the same system state more than once and exit it in different directions.[8] Our encoding forbids that.[9] I.e., in any system satisfying the derived LTL formula, a system path mapped out by an ID has a unique outgoing direction from every visited state. As a consequence, such systems are less concise. This is illustrated in the following example.

[7] To see this, recall that we can get UCW_{ψ} by treating $NBW_{\neg\psi}$ as a UCW, and notice that $|NBW_{\mathsf{F}[p\wedge\mathsf{G}d\wedge\neg\varphi]}| = |NBW_{\neg\varphi}| + 1$.

[8] E.g., in Figure 4.3a the system path $t_0t_1t_1(t_0)^{\omega}$, satisfying $\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$, double-visits state t_1 and exits it first in direction r and then in $\neg r$, where t_0 is the system state on the left and t_1 is on the right.

[9] Recall that with $\mathsf{E}\varphi$ we associate a number $v_{\mathsf{E}\varphi}$, such that whenever in a system state $v_{\mathsf{E}\varphi}$ is non-zero, then the path mapped out by $v_{\mathsf{E}\varphi}$-numbered directions satisfies the path formula φ. Therefore whenever $v_{\mathsf{E}\varphi}$-numbered path visits a system state, it exits it in the *same* direction $d_{v_{\mathsf{E}\varphi}}$.

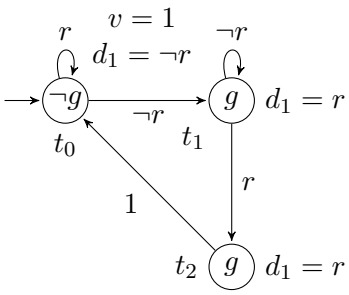


Figure 4.7: A smallest Moore machine satisfying the LTL formula from Example 8.

Example 8 (Non-minimality). Let $I = \{r\}$, $O = \{g\}$, and consider the CTL* formula

$$\mathsf{EX}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g))$$

The NBW automaton for the path formula has 5 states (Figure 4.1a), so we introduce the integer proposition v ranging over $\{0, ..., 5\}$ and Boolean propositions d_1, d_2, d_3, d_4, d_5. The LTL formula is

$$v \neq 0 \;\wedge \bigwedge_{j \in \{1...5\}} \mathsf{G}\left[v = j \;\rightarrow\; (\mathsf{G}\,d_j \rightarrow \mathsf{X}(g \wedge \mathsf{X}(g \wedge \mathsf{F}\,\neg g)))\right]$$

A smallest system for this LTL formula is in Figure 4.7. It is of size is 3, while a smallest system for the original CTL* formula is of size 2 (Figure 4.3a).

4.2.3 Bounded Reduction

While we have realisability equivalence for sufficiently large k, k is a parameter, where much smaller k might suffice. In the spirit of Bounded Synthesis, it is possible to use smaller parameters in the hope of finding a system. These systems might be of interest in that they guarantee a limited entanglement of different mapped out paths, as they cap the number of such paths that can go through the same node of a tree. Such systems are therefore simple wrt. this metric, and this metric is independent of the automaton representation. (As opposed to a lower bound for k that depends on the existential automata.)

4.3 Checking Unrealisability of CTL*

What does a witness of unrealisability for CTL* look like? I.e., when a formula is unrealisable, is there an "environment model", like in the LTL case, which disproves any system model?

The LTL formula and the annotation shed light on this: the system for the dualised case is a strategy how to choose original inputs (depending on the history of v, d, p, and original outputs), such that any path in the resulting tree violates the original LTL formula. I.e., the spoiler strategy is a tree, whose nodes are labeled with original inputs, and whose directions are defined by v, d, p, and original outputs.

Example 9. Consider an unrealisable CTL* specification: $\mathsf{AG}\, g \wedge \mathsf{EFX}\, \neg g$ with inputs $\{r\}$ and outputs $\{g\}$. After reduction to LTL we get the specification: inputs $\{r\}$, outputs $\{g, p_{\mathsf{AG}g}, v_{\mathsf{EFX}\bar{g}}, d_1, d_2\}$, and the LTL formula

$$p_{\mathsf{AG}g} \wedge v_{\mathsf{EFX}\bar{g}} \neq 0 \wedge \mathsf{G}\left[p_{\mathsf{AG}g} \rightarrow \mathsf{G}\, g\right] \wedge \bigwedge_{j \in \{1,2\}} \mathsf{G}\left[(v_{\mathsf{EFX}\bar{g}} = j \wedge \mathsf{G}\, d_j) \rightarrow \mathsf{FX}\, \neg g\right].$$

The dual specification is: the system type is Mealy, new inputs $\{g, p_{\mathsf{AG}g}, v_{\mathsf{EFX}\bar{g}}, d_1, d_2\}$, new outputs $\{r\}$, and the LTL formula is the negated original LTL:

$$p_{\mathsf{AG}g} \wedge v_{\mathsf{EFX}\bar{g}} \neq 0 \wedge \mathsf{G}\left[p_{\mathsf{AG}g} \rightarrow \mathsf{G}\, g\right] \;\rightarrow \bigvee_{j \in \{1,2\}} \mathsf{F}\left[(v_{\mathsf{EFX}\bar{g}} = j \wedge \mathsf{G}\, d_j) \wedge \mathsf{GX}\, g\right].$$

This dual specification is realisable, and it exhibits, e.g., the following witness of unrealisability: the output r follows d_1 or d_2 depending on input $v_{\mathsf{EFX}\bar{g}}$. (The new system needs two states. State 1 describes "I've seen $v_{\mathsf{EFX}\bar{g}} \in \{0, 1\}$ and I output r equal to d_1"; from state 1 we irrevocably go into state 2 once $v_{\mathsf{EFX}\bar{g}} = 2$ and make r equal to d_2).

Although our encoding allows for checking unrealisability of CTL* (via dualising the converted LTL specification), this approach suffers from a very high complexity. Recall that the LTL formula can become exponential in the size of a CTL* formula, which could only be handled because it became a big conjunction with sufficiently small conjuncts. After negating it becomes a large disjunction, which makes the corresponding UCW doubly exponential in the size of the initial CTL* specification (vs. single exponential for the non-negated case). This seems—there may be a more clever analysis of the formula structure—to make the unrealisability check via reduction to LTL cost three exponents in the worst case (vs. 2EXP by the standard approach).

What one could try is to let the new system player in the dualised game choose a number of disjunctive formulas to follow, and allow it to revoke the choice finitely many times. This is conservative: if following m different disjuncts in the dualised formula is enough to win, then the new system wins.

Alternatively, one could try to synthesise environment model for parts of the disjunction increasing them until all disjunctions are used. This is precise.

4.4 Experiments

We implemented the CTL* to LTL converter `ctl_to_ltl.py` inside PARTY [57]. PARTY also has two implementations of Bounded Synthesis [46], one encodes the problem into SMT and another reduces the problem to safety games. Also, PARTY has a CTL* synthesiser based on Bounded Synthesis idea that encodes the problem into SMT (presented in Chapter 3). In this section we compare those three solvers, where the first two solvers take LTL formulas produced by our converter. All logs and the code are available in repository https://github.com/5nizza/party-elli, the branch "cav17".

Specifications. We created realisable arbiter-like CTL* specifications. The number after the specification name indicates the number of clients. All specifications

have LTL properties in the spirit of "every request must eventually be granted" and the mutual exclusion of the grants, plus some CTL* properties. Below we provide details.

- "res_arbiter" has the properties:
  $\bigwedge_{i\neq j} \mathsf{AG}\,\neg(g_i \wedge g_j) \;\wedge\; \bigwedge_i \mathsf{AG}(r_i \rightarrow \mathsf{F}\,g_i) \wedge$
  $\bigwedge_i \mathsf{AG}\,\mathsf{EFG}(\neg g_i)$.

- "loop_arbiter" has the properties:
  $\bigwedge_{i\neq j} \mathsf{AG}\,\neg(g_i \wedge g_j) \;\wedge\; \bigwedge_i \mathsf{AG}(r_i \rightarrow \mathsf{F}\,g_i) \wedge$
  $\bigwedge_i \mathsf{AG}\,\mathsf{EFG}(\neg g_i) \wedge$
  $\bigwedge_i \mathsf{EFG}\,g_i$.

- "postp_arbiter" has the properties:
  $\bigwedge_{i\neq j} \mathsf{AG}\,\neg(g_i \wedge g_j) \;\wedge\; \bigwedge_i \mathsf{AG}(r_i \rightarrow \mathsf{F}\,g_i) \wedge$
  $\bigwedge_i \neg g_i \wedge$
  $\bigwedge_i \mathsf{AGEF}(\neg g_i \wedge r_i \wedge \mathsf{X}(\neg g_i \wedge r_i \wedge X\neg g_i))$.

- "prio_arbiter" has the properties:
  $\mathsf{A}\big[\,\mathsf{GF}\,\neg rm \rightarrow$
  $\quad \bigwedge_{i\neq j} \mathsf{G}\,\neg(g_i \wedge g_j) \wedge \mathsf{G}\,\neg(g_i \wedge gm) \wedge \bigwedge_i \mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i) \wedge \mathsf{G}(rm \rightarrow \mathsf{F}\,gm) \wedge$
  $\quad \mathsf{G}(rm \rightarrow \mathsf{X}(\bigwedge_i \neg g_i\,\mathsf{U}\,gm))\big] \wedge$
  $\bigwedge_i \mathsf{AG}\,\mathsf{EFG}(\neg g_i) \wedge \mathsf{AG}\,\mathsf{EFG}(\neg gm)$.
  (It additionally has the prioritised request input rm and grant output gm.)

- "user_arbiter" contains only existential properties that specify different sequences of requests and grants.

LTL formula and automata sizes. Our experiments confirm that the LTL formulas increase $\approx |Q|$ times when k increases from 1 to $|Q|$, just as described by Eq. 4.1. But the increase does not incur the exponential blow up of the UCWs: they also increase only $\approx |Q|$ times (just like the theory predicts).

Synthesis time. The table below compares different synthesis approaches for the (realisable) CTL* specifications described above. The column |CTL*| is the size of the non-reduced AST of the CTL* formula, the column |LTL| has two numbers: the size of the non-reduced AST of the LTL formula for $k = 1$ (k is the number of witness IDs) and the size for k being the upper bound (the sum of the number of states in all existential automata). The column $|AHT|$ is the sum of the number of states in existential and universal automata. The column $|UCW|$ is the number of states in the UCW of the translated LTL formula: we show two numbers, for $k = 1$ and when it is the upper bound. Timings are in seconds, the timeout is 3 hours[10] (denoted "*to*"). "Time CTL*" is the synthesis time and [system size] required for CTL* synthesizer `star.py`, "time LTL(SMT)" — for synthesizer `elli.py` which implements the original Bounded Synthesis for LTL via SMT [46], "time LTL(game)" — for synthesizer `kid.py` which implements the original Bounded Synthesis for LTL via reduction to safety games [46]. Both

[10]Except for the last specification "user_arbiter1" for which the timeout was 1 hour.

"time LTL" columns have two numbers: when k is set to the minimal value for which the LTL is realisable, and when k is set to the upper bound. The subscript near the number indicates the value of k: e.g., to_8 means the timeout on all values of k from 1 to $|Q| = 8$; $to_{12(3)}$ means there was the timeout for $k = |Q| = 12$ and the last non-timeout was for $k = 3$; 20_1 means 20 seconds and the minimal k is 1. The running commands were: "`elli.py --incr spec`", "`star.py --incr spec`", "`kid.py spec`".

	$\|\text{CTL}^*\|$	$\|\text{LTL}\|$ $(k_1{:}k_{\|Q\|})$	$\|\text{AHT}\|$	$\|\text{UCW}\|$ $(k_1{:}k_{\|Q\|})$	time CTL^*	time LTL(SMT) $(k_{min}{:}k_{\|Q\|})$	time LTL(game) $(k_{min}{:}k_{\|Q\|})$
res_arbiter3	65	78 : 127	9	7 : 9	25 [5]	$40_1 : 260_2$	$\mathbf{7_1 : 20_2}$
res_arbiter4	97	109 : 168	10	8 : 10	7380 [7]	to_1	$\mathbf{30_1 : 60_2}$
loop_arbiter2	49	105 : 682	12	11 : 41	**2** [4]	$20_3 : 131_6$	$18_3 : to_{6(5)}$
loop_arbiter3	80	183 : 1607	15	14 : 70	**6360** [7]	to_8	to_8
postp_arbiter3	113	177 : 2097	19	15 : 114	3 [4]	$\mathbf{2_1} : 1735_{12}$	$20_1 : to_{12(3)}$
postp_arbiter4	162	276 : 4484	24	19 : *to*	2920 [5]	$\mathbf{68_1} : to_{16(5)}$	$70_1 : to_{16(2)}$
prio_arbiter2	82	92 : 141	13	14 : 16	60 [5]	$14_1 : 19_2$	$\mathbf{9_1 : 17_2}$
prio_arbiter3	117	125 : 184	15	16 : 18	*to*	$4318_1 : to_2$	$\mathbf{26_1 : 56_2}$
user_arbiter1	99	203 : 4323	23	23 : *to*	**3** [5]	to_{16}	to_{16}

When the minimal k is 1, the game-based synthesiser is the fastest in most of the cases. However, it struggles to find a system when we set k to a "large" number (see the timeouts in rows 3–6). The LTL part of specifications "res_arbiter" and "prio_arbiter" is known to be easier for the game-based synthesiser than for the SMT-based ones—adding the simple resettability property does not change this. For CTL* specifications whose minimal k is "large" ("loop_arbiter" and "user_arbiter" that requires $k > 4$), the specialised CTL* synthesiser outperforms both the game-based and SMT-based synthesisers for the translated LTL specifications. Our preliminary conclusion is that for CTL* specifications that do not require large k, the reduction to LTL synthesis is beneficial. (Currently we do not know how to predict if a large k is required.)

System sizes. The reduction did not increase the system size in most of the cases (for the cases "loop_arbiter3", "res_arbiter4", and "user_arbiter1" we do not know the minimal system size when synthesising from the LTL specification).

4.5 Conclusion

We presented the reduction of CTL* synthesis problem to LTL synthesis problem. The reduction preserves the worst-case complexity of the synthesis problem, although possibly at the cost of larger systems. The reduction allows the designer to write CTL* specifications even when she has only an LTL synthesiser at hand. We experimentally showed—on the *small* set of specifications—that the reduction is practical when the number of existentially quantified formulas is small.

We briefly discussed how to handle unrealisable CTL* specifications. Whether our suggestions are practical on typical specifications—this is still an open question. A possible future direction is to develop a similar reduction for logics like ATL* [2], and to look into the problem of satisfiability of CTL* [43].

Part II

Excursion Into Parameterized Systems

Guarded and Token-ring Systems

Overview of Part II

Concurrent systems are hard to implement and even harder to debug. On the other side, they are relatively easy to specify. Consider, for example, the arbiter serving many clients. A possible specification is $\forall i \neq j.\, \mathsf{G}\, \neg(g_i \wedge g_j) \wedge \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i)$, which says that, for every client, every request should be eventually granted, and the grants are mutually exclusive. If a human implements such an arbiter, he would try to come up with a basic block that handles a single client, and connect such a block into a system, that handles as many clients as needed. On the other side, the computer tries to synthesize a system as one monolithic block. This hides the insight that a system for $n+1$ clients is very similar to a system for n clients. This leads to the scalability problem, once we require a large number of clients.

The parameterized synthesis approach [50] addresses the issue. The idea is—just like the human would do—to automatically synthesize a basic block that can be arranged into a system of any desired size. There are several ways to arrange such blocks into a system, depending on how they communicate with each other. In this thesis part we will look into two system architectures.

The first architecture is inspired by cache coherence protocols found in modern processors. Such a protocol is described by states, where transitions between states happen depending on whether or not there is a processor in a particular state. I.e., the transitions are guarded. Chapter 5 studies guarded systems.

The second kind of systems is token-ring systems. In such a system, the single token circulates in the system. A process possessing the token knows that no other process has the token. Based on this information, the process can, for example, raise the grant. If all processes raise the grant only when they posses the token, then the grants will be mutually exclusive. Chapter 6 studies token-ring systems.

For both architectures we study their parameterized synthesis problems. The parameterized synthesis problems asks, given a parameterized specification, to find a process implementation, such that a system of any size composed of such processes, satisfies the specification. The solution to the seemingly difficult problem—we now ask for correctness of a system of *any* size—is based on the cutoff reduction: to synthesize a process that works for all system sizes, it is enough to synthesize a process that works in a system of a cutoff size. For example, for the specification of the arbiter mentioned above, the cutoff for token-ring systems is 4. This means that it is enough to find a process implementation that works in a system with 4 such processes. Once we find it, a system of size 5, 6, 7,... is also correct.

In Chapter 5 we prove cutoff results for guarded systems. Our results extend the results of Emerson and Kahlon [38]. Our contribution concerns both parameterized synthesis and parameterized verification. We prove new cutoff results that are applicable to a previously unconsidered setting of open systems with liveness properties under fairness assumptions. We also prove new cutoff results for deadlock detection. The work is theoretical; it is yet to find its application.

In Chapter 6 on token-ring systems, we extend the cutoffs of Emerson and Namjoshi [40] to a new setting of fully asynchronous systems and richer specifications. Then we apply them to an industrial arbiter protocol called AMBA. Thus, we synthesize for the first time the AMBA protocol in the parameterized sense.

The chapters can be read in any order.

Chapter 5

Parameterized Guarded Systems

This chapter is based on joint work with S.Außerlechner and S.Jacobs [7, 6]

Abstract. Guarded protocols were introduced in a seminal paper by Emerson and Kahlon (2000), and describe systems of processes whose transitions are enabled or disabled depending on the existence of other processes in certain local states. In this chapter we study parameterized model checking and synthesis of guarded protocols, both aiming at formal correctness arguments for systems with any number of processes. Cutoff results reduce reasoning about systems with an arbitrary number of processes to systems of a determined, fixed size. Our work stems from the observation that existing cutoff results for guarded protocols (i) are restricted to closed systems, and (ii) are of limited use for liveness properties because reductions do not preserve fairness. We close these gaps and obtain new cutoff results for open systems with liveness properties under fairness assumptions. Furthermore, we obtain cutoffs for the detection of global and local deadlocks, which are of paramount importance in synthesis. Finally, we prove tightness or asymptotic tightness for the new cutoffs.

5.1 Introduction

Concurrent hardware and software systems are notoriously hard to get right. Formal methods like model checking or synthesis can be used to guarantee correctness, but the state explosion problem prevents us from using such methods for systems with a large number of components. Furthermore, correctness properties are often expected to hold for an *arbitrary* number of components. Both problems can be solved by *parameterized* model checking and synthesis approaches, which give correctness guarantees for systems with any number of components without considering every possible system instance explicitly.

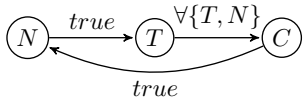
While parameterized model checking (PMC) is undecidable in general [79], there exists a number of methods that decide the problem for specific classes of systems [47, 37, 40], as well as semi-decision procedures that are successful in many interesting cases [64, 29, 53]. In this chapter, we consider the *cutoff* method

that can guarantee properties of systems of arbitrary size by considering only systems of up to a certain fixed size, thus providing a decision procedure for PMC if components are finite-state.

We consider systems that are composed of an arbitrary number of processes, each an instance of a process template from a given, finite set. Process templates can be viewed as synchronization skeletons [36], i.e., program abstractions that suppress information not necessary for synchronization. In our system model, processes communicate by guarded updates, where guards are statements about other processes that are interpreted either conjunctively ("every other process satisfies the guard") or disjunctively ("there exists a process that satisfies the guard"). Conjunctive guards can model atomic sections or locks, disjunctive guards can model token-passing or to some extent pairwise rendezvous (cf. [38]).

This class of systems has been studied by Emerson and Kahlon [37], and cutoffs that depend on the size of process templates are known for specifications of the form $\forall\bar{p}.\ \Phi(\bar{p})$, where $\Phi(\bar{p})$ is an LTL\X property over the local states of one or more processes $\bar{p}$. Note that this does not allow us to specify fairness assumptions, for two reasons: (i) to specify fairness, additional atomic propositions for enabledness and scheduling of processes are needed, and (ii) specifications with global fairness assumptions are of the form $(\forall\bar{p}.\ \mathit{fair}(\bar{p})) \rightarrow (\forall\bar{p}.\ \Phi(\bar{p}))$. Because neither is supported by [37], the existing cutoffs are of limited use for reasoning about liveness properties.

Emerson and Kahlon [37] mentioned this limitation and illustrated it using the process template on the figure on the right. Transitions from the initial state N to the "trying" state T, and from the critical state C to N are always possible, while the transition from T to C is only possible if no other process is in C. The existing cutoff results can be used to prove safety properties like mutual exclusion for systems composed of arbitrarily many copies of this template. However, they cannot be used to prove starvation-freedom properties like $\forall p.\, \mathsf{A}\,\mathsf{G}(T_p \rightarrow \mathsf{F}\, C_p)$, stating that every process p that enters its local state T_p will eventually enter state C_p, because without fairness of scheduling the property does not hold.



Also, Emerson and Kahlon [37] consider only closed systems. Therefore, in this example, processes always try to enter C. In contrast, in open systems the transition to T might be a reaction to a corresponding input from the environment that makes entering C necessary. While it is possible to convert an open system to a closed system that is equivalent under LTL properties, this comes at the cost of a blow-up.

Motivation. Our work is inspired by applications in parameterized synthesis [50], where the goal is to automatically construct process templates such that a given specification is satisfied in systems with an arbitrary number of components. In this setting, one generally considers *open systems* that interact with an uncontrollable environment (user). Also, most specifications contain liveness properties that cannot be guaranteed without fairness assumptions. Note that in the parameterized setting liveness properties cannot be reduced to safety properties, because the size of a system is not bounded a priori. Finally, we are interested in synthesizing

deadlock-free systems. *Cutoffs* are essential for parameterized synthesis, because they enable a semi-decision procedure to parameterized synthesis.

Contributions.

- We show that existing cutoffs for model checking of LTL\X properties are in general not sufficient for systems with *fairness assumptions*, and provide new cutoffs for this case.
- We improve some of the existing cutoff results, and give separate cutoffs for the problem of *deadlock detection*, which is closely related to fairness.
- We prove *tightness* or asymptotical tightness for all of our cutoffs, showing that smaller cutoffs cannot exist with respect to the parameters we consider.

Moreover, all of our cutoffs directly support *open systems*, where each process may communicate with an adversarial environment. This makes the blow-up incurred by translation to an equivalent closed system unnecessary. Finally, we will show in Sect. 5.4 how to integrate our size-dependent cutoffs into the parameterized synthesis approach.

5.2 Related Work

In this work we extend the results of Emerson and Kahlon [37] who study PMC of guarded protocols, but do not support fairness assumptions, nor provide cutoffs for deadlock detection. In [38] they extended their work to systems with limited forms of guards and broadcasts, and also proved undecidability of PMC of conjunctive guarded protocols wrt. LTL (including X), and undecidability wrt. LTL\X for systems with both conjunctive and disjunctive guards.

Bouajjani et al. [24] study parameterized model checking of resource allocation systems (RASs). Such systems have a bounded number of resources, each owned by at most one process at any time. Processes are pushdown automata, and can request resources with high or normal priority. RASs are similar to conjunctive guarded protocols in that certain transitions are disabled unless a processes has a certain resource. RASs without priorities and where all the processes are finite state Moore machines can be converted to conjunctive guarded protocols (at the price of blow up), but not vice versa. The authors study parameterized model checking wrt. LTL\X properties under certain fairness assumptions, and deadlock detection. Their proofs are based on ideas of [37] (our proofs are also based on ideas of [37]).

German and Sistla [47] considered global deadlocks and strong fairness properties for systems with pairwise rendezvous communication in a clique. In such systems, processes communicate pairwise using messages: one process sends a message and blocks until another process reads the message. Emerson and Kahlon [38] have shown that disjunctive guard systems can be reduced to such pairwise rendezvous systems. However, German and Sistla [47] do not provide cutoffs, nor do they consider deadlocks for individual processes, and their specifications can talk about one process only. Aminof et al. [4] have recently extended these results to

more general topologies, and have shown that for some decidable parameterized model checking problems there are no cutoffs, even in cliques.

Many of the decidability results above have been surveyed in our book [21].

5.3 Preliminaries

Many definitions intersect with those defined in previous chapters, but to keep the chapter self-contained we define them here.

Notation: $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ is the set of Boolean values, $\mathbb{N}$ is the set of natural numbers (excluding 0), $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, $[k]$ is the set $\{i \in \mathbb{N} \mid i \leq k\}$ and $[0..k]$ is the set $[k] \cup \{0\}$ for $k \in \mathbb{N}$. For a sequence $x = x_1 x_2 \ldots$ denote the i-j-subsequence as $x[i\!:\!j]$, i.e., $x[i\!:\!j] = x_i \ldots x_j$.

5.3.1 System Model

We consider systems $A \| B^n$, usually written $(A, B)^{(1,n)}$, consisting of one copy of a process template A and n copies of a process template B, in an interleaving parallel composition.We distinguish objects that belong to different templates by indexing them with the template. E.g., for process template $U \in \{A, B\}$, Q_U is the set of states of U. For this section, fix two disjoint finite sets Q_A, Q_B as sets of states of process templates A and B, and a positive integer n.

Processes. A *process template* is a transition system $U = (Q, \mathsf{init}, \Sigma, \delta)$ with

- Q is a finite set of states including the initial state init,
- Σ is a finite input alphabet,
- $\delta : Q \times \Sigma \times \mathcal{P}(Q_A \dot\cup Q_B) \times Q$ is a guarded transition relation.

A process template is *closed* if $\Sigma = \emptyset$, and otherwise *open*.

By $q_i \xrightarrow{e:g} q_j$ we denote a process transition from q_i to q_j for input $e \in \Sigma$ and guarded by guard $g \in \mathcal{P}(Q_A \dot\cup Q_B)$. We skip the input e and guard g if they are not important or can be inferred from the context.

We define the size $|U|$ of a process template $U \in \{A, B\}$ as $|Q_U|$. A copy of a template U will be called a *U-process*. Different B-processes are distinguished by subscript, i.e., for $i \in [1..n]$, B_i is the ith copy of B, and q_{B_i} is a state of B_i. A state of the A-process is denoted by q_A.

For the rest of this subsection, fix templates A and B. We assume that $\Sigma_A \cap \Sigma_B = \emptyset$. We will also write p for a process in $\{A, B_1, \ldots, B_n\}$, unless p is specified explicitly. We often denote the set $\{B_1, ..., B_n\}$ as $\mathcal{B}$.

Disjunctive and conjunctive systems. In a system $(A, B)^{(1,n)}$, consider the global state $s = (q_A, q_{B_1}, \ldots, q_{B_n})$ and global input $e = (\sigma_A, \sigma_{B_1}, \ldots, \sigma_{B_n})$. We write $s(p)$ for q_p, and $e(p)$ for σ_p. A local transition $(q_p, \sigma_p, g, q'_p) \in \delta_U$ of a process p is *enabled for s and e* if the *guard* g is satisfied by the state s wrt. the process p, written $(s, p) \models g$ (defined below). The semantics of $(s, p) \models g$ differs for

disjunctive and conjunctive systems:

$$\begin{aligned}
\text{In disjunctive systems: } (s,p) \models g \quad &\text{iff} \quad \exists p' \in \{A, B_1, \ldots, B_n\} \setminus \{p\} : \quad q_{p'} \in g. \\
\text{In conjunctive systems: } (s,p) \models g \quad &\text{iff} \quad \forall p' \in \{A, B_1, \ldots, B_n\} \setminus \{p\} : \quad q_{p'} \in g.
\end{aligned}$$

Note that we check containment in the guard (disjunctively or conjunctively) only for local states of processes *different from* p. A process is *enabled* for s and e if at least one of its transitions is enabled for s and e, otherwise it is *disabled*.

Like Emerson and Kahlon [37], we assume that in conjunctive systems init_A and init_B are contained in all guards, i.e., they act as neutral states. Furthermore, we call a conjunctive system 1-*conjunctive* if every guard is of the form $(Q_A \dot\cup Q_B) \setminus \{q\}$ for some $q \in Q_A \dot\cup Q_B$.

Then, $(A,B)^{(1,n)}$ is defined as the transition system $(S, \mathsf{init}_S, E, \delta)$ with

- set of global states $S = Q_A \times Q_B^n$,
- global initial state $\mathsf{init}_S = (\mathsf{init}_A, \mathsf{init}_B, \ldots, \mathsf{init}_B)$,
- set of global inputs $E = (\Sigma_A) \times (\Sigma_B)^n$,
- and global transition relation $\delta \subseteq S \times E \times S$ with $(s,e,s') \in \delta$ iff
  - i) $s = (q_A, q_{B_1}, \ldots, q_{B_n})$,
  - ii) $e = (\sigma_A, \sigma_{B_1}, \ldots, \sigma_{B_n})$, and
  - iii) s' is obtained from s by replacing one local state q_p with a new local state q'_p, where p is a U-process with local transition $(q_p, \sigma_p, g, q'_p) \in \delta_U$ and $(s,p) \models g$. Thus, we consider so-called interleaved systems, where in each step exactly one process transits.

We say that a system $(A,B)^{(1,n)}$ is *of type* (A,B). It is called a *conjunctive system* if guards are interpreted conjunctively, and a *disjunctive system* if guards are interpreted disjunctively. A system is *closed* if all of its templates are closed.

Runs. A *configuration* of a system is a triple (s,e,p), where $s \in S$, $e \in E$, and p is either a system process, or the special symbol $\perp$. A *path* of a system is a configuration sequence $x = (s_1,e_1,p_1),(s_2,e_2,p_2),\ldots$ such that, for all $m < |x|$, there is a transition $(s_m, e_m, s_{m+1}) \in \delta$ based on a local transition of process p_m. We say that process p_m *moves* at *moment* m. Configuration $(s,e,\perp)$ appears iff all processes are disabled for s and e. Also, for every p and $m < |x|$: either $e_{m+1}(p) = e_m(p)$ or process p moves at moment m. That is, the environment keeps the input to each process unchanged until the process can read it.[1]

A system *run* is a maximal path starting in the initial state. Runs are either infinite, or they end in a configuration $(s,e,\perp)$. We say that a run is *initializing* if every process that moves infinitely often also visits its init infinitely often.

[1]By only considering inputs that are actually processed, we approximate an action-based semantics. Paths that do not fulfill this requirement are not very interesting, since the environment can violate any interesting specification that involves input signals by manipulating them when the corresponding process is not allowed to move.

Given a system path $x = (s_1, e_1, p_1), (s_2, e_2, p_2), \ldots$ and a process p, the *local path* of p in x is the projection $x(p) = (s_1(p), e_1(p)), (s_2(p), e_2(p)), \ldots$ of x onto local states and inputs of p. Similarly, we define the projection on two processes p_1, p_2 denoted by $x(p_1, p_2)$.

Deadlocks and fairness. A run is *globally deadlocked* if it is finite. An infinite run is *locally deadlocked* for process p if there exists m such that p is disabled for all $s_{m'}, e_{m'}$ with $m' \geq m$. A run is *deadlocked* if it is locally or globally deadlocked. A system *has a (local/global) deadlock* if it has a (locally/globally) deadlocked run. Note that the absence of local deadlocks for all p implies the absence of global deadlocks, but not the other way around.

A run $(s_1, e_1, p_1), (s_2, e_2, p_2), ...$ is *unconditionally-fair* if every process moves infinitely often. A run is *strong-fair* if it is infinite and, for every process p, if p is enabled infinitely often, then p moves infinitely often. We will discuss the role of deadlocks and fairness in synthesis in Section 5.4.

Remark 8 ($A^m \| B^n$)**.** One usually starts with studying parameterized systems of the form A^n (having one process template), then proceeds to systems of the form $A^m \| B^n$ (having two templates) and $U_1^{n_1} \| \ldots \| U_m^{n_m}$ (having an arbitrary fixed number of templates). Our work studies systems $A \| B^n$, which have one A-process and a parameterized number of B-processes, because the results for such systems can be generalized to systems $U_1^{n_1} \| \ldots \| U_m^{n_m}$ (see [37] for details). This generalization works for our results as well, except for the cutoffs for deadlock detection that are restricted to 1-conjunctive systems of the form $A \parallel B^n$ (Section 5.5).

5.3.2 Specifications

Fix templates (A, B). We consider formulas in LTL\X—LTL without the next-time operator X—that are prefixed by path quantifiers E or A (for LTL and path quantifiers see Section 2.3). Let $h(A, B_{i_1}, \ldots, B_{i_k})$ be an LTL\X formula over atomic propositions from $Q_A \cup \Sigma_A$ and indexed propositions from $(Q_B \cup \Sigma_B) \times \{i_1, \ldots, i_k\}$. For a system $(A, B)^{(1,n)}$ with $n \geq k$ and every $i_j \in [1..n]$, satisfaction of $\mathsf{A}h(A, B_{i_1}, \ldots, B_{i_k})$ and $\mathsf{E}h(A, B_{i_1}, \ldots, B_{i_k})$ is defined in the usual way.

Parameterized specifications. A *parameterized specification* is a temporal logic formula with indexed atomic propositions and quantification over indices. We consider formulas of the forms $\forall i_1, \ldots, i_k.\mathsf{A}h(A, B_{i_1}, \ldots, B_{i_k})$ and $\forall i_1, \ldots, i_k.\mathsf{E}h(A, B_{i_1}, \ldots, B_{i_k})$. For a given $n \geq k$,

$$(A, B)^{(1,n)} \models \forall i_1, \ldots, i_k.\mathsf{A}h(A, B_{i_1}, \ldots, B_{i_k})$$

iff

$$(A, B)^{(1,n)} \models \bigwedge_{j_1 \neq \ldots \neq j_k \in [1..n]} \mathsf{A}h(A, B_{j_1}, \ldots, B_{j_k}).$$

By symmetry of guarded systems (see [37]), the second formula is equivalent to $(A, B)^{(1,n)} \models \mathsf{A}h(A, B_1, \ldots, B_k)$. The formula $\mathsf{A}h(A, B_1, \ldots, B_k)$ is denoted by $\mathsf{A}h(A, B^{(k)})$, and we often use it instead of the original $\forall i_1, \ldots, i_k.\mathsf{A}h(A, B_{i_1}, ..., B_{i_k})$. For formulas with the path quantifier E, satisfaction is defined analogously and is equivalent to satisfaction of $\mathsf{E}h(A, B^{(k)})$.

Example 10. Consider the formula

$$\forall i_1, i_2.\mathsf{A}\big(\,\mathsf{G}(r_{i_1} \to \mathsf{F}\, g_{i_1}) \wedge \mathsf{G}\,\neg(g_{i_1} \wedge g_{i_2})\big).$$

By our definition, its satisfaction by a system $(A, B)^{(1,3)}$ means

$$(A, B)^{(1,3)} \models \mathsf{A}\left(\begin{array}{l}\mathsf{G}(r_1 \to \mathsf{F}\, g_1) \wedge \mathsf{G}(r_2 \to \mathsf{F}\, g_2) \wedge \mathsf{G}(r_3 \to \mathsf{F}\, g_3) \wedge \\ \mathsf{G}\,\neg(g_1 \wedge g_2) \wedge \mathsf{G}\,\neg(g_1 \wedge g_3) \wedge \mathsf{G}\,\neg(g_2 \wedge g_3)\end{array}\right),$$

where g_1 and r_1 refer to the propositions g and r of the process B_1, g_2 and r_2 belong to B_2, and so on. By symmetry, the latter satisfaction is equivalent to

$$(A, B)^{(1,3)} \models \mathsf{A}\left(\begin{array}{l}\mathsf{G}(r_1 \to \mathsf{F}\, g_1) \wedge \mathsf{G}(r_2 \to \mathsf{F}\, g_2) \wedge \\ \mathsf{G}\,\neg(g_1 \wedge g_2)\end{array}\right).$$

Note that this formula talks about processes B_1 and B_2, but does not mention B_3.

Specification of fairness and local deadlocks. It is often convenient to express fairness assumptions and local deadlocks as parameterized specifications. To this end, define auxiliary atomic propositions move_p and en_p for every process p of system $(A, B)^{(1,n)}$. At moment m of a given run $(s_1, e_1, p_1), (s_2, e_2, p_2), \ldots$, let move_p be true whenever $p_m = p$, and let en_p be true if p is enabled for s_m, e_m. Note that we only allow the use of these propositions to define fairness, but not in general specifications. Then, an infinite run is

- *local-deadlock-free* if it satisfies $\forall p.\, \mathsf{GF}\, \mathsf{en}_p$, abbreviated as $\Phi_{\neg dead}$,
- *strong-fair* if it satisfies $\forall p.\, \mathsf{GF}\, \mathsf{en}_p \to \mathsf{GF}\, \mathsf{move}_p$, abbreviated as Φ_{strong}, and
- *unconditionally-fair* if it satisfies $\forall p.\, \mathsf{GF}\, \mathsf{move}_p$, abbreviated as Φ_{uncond}.

If $f \in \{strong, uncond\}$ is a fairness notion and $\mathsf{A}h(A, B^{(k)})$ a specification, then we write $\mathsf{A}_f h(A, B^{(k)})$ for $\mathsf{A}(\Phi_f \to h(A, B^{(k)}))$. Similarly, we write $\mathsf{E}_f h(A, B^{(k)})$ for $\mathsf{E}(\Phi_f \wedge h(A, B^{(k)}))$.

5.3.3 Model Checking and Synthesis Problems

Given a system $(A, B)^{(1,n)}$ and a specification $\mathsf{A}h(A, B^{(k)})$, where $n \geq k$. Then:

- the *model checking problem* is to decide whether $(A, B)^{(1,n)} \models \mathsf{A}h(A, B^{(k)})$,
- the *deadlock detection problem* is to decide whether $(A, B)^{(1,n)}$ does not have global nor local deadlocks,
- the *parameterized model checking problem* (PMCP) is to decide whether $\forall m \geq n:\ (A, B)^{(1,m)} \models \mathsf{A}h(A, B^{(k)})$, and
- the *parameterized deadlock detection problem* is to decide whether, for all $m \geq n$, $(A, B)^{(1,m)}$ does not have global nor local deadlocks.

For a given number $n \in \mathbb{N}$ and specification $\mathsf{A}h(A, B^{(k)})$ with $n \geq k$,

- the *template synthesis problem* is to find process templates A, B such that $(A,B)^{(1,n)} \models \mathsf{A}h(A, B^{(k)})$ and $(A,B)^{(1,n)}$ does not have global deadlocks[2]
- the *bounded template synthesis problem* for a pair of bounds $(b_A, b_B) \in \mathbb{N} \times \mathbb{N}$ is to solve the template synthesis problem with $|A| \leq b_A$ and $|B| \leq b_B$.
- the *parameterized template synthesis problem* is to find process templates A, B such that $\forall m \geq n :\ (A,B)^{(1,m)} \models \mathsf{A}h(A, B^{(k)})$ and $(A,B)^{(1,m)}$ does not have global deadlocks[2].

Similarly, we define problems for specifications having E instead of A. The definitions can be flavored with different notions of fairness.

5.4 Reduction Method and Challenges

We show how to use existing cutoff results of Emerson and Kahlon [37] to reduce the PMCP to a standard model checking problem, and parameterized template synthesis to template synthesis. We note the limitations of the existing results that are crucial in the context of synthesis.

Reduction by Cutoffs

Cutoffs. A *cutoff* for a system type (A, B) and a specification Φ is a number $c \in \mathbb{N}$ such that:

$$\forall n \geq c : \left((A,B)^{(1,c)} \models \Phi \quad \Leftrightarrow \quad (A,B)^{(1,n)} \models \Phi\right).$$

Similarly, a *cutoff for deadlock detection* for a system type (A, B) is a number $c \in \mathbb{N}$ such that:

$$\forall n \geq c : \left((A,B)^{(1,c)} \textit{ has a deadlock} \quad \Leftrightarrow \quad (A,B)^{(1,n)} \textit{ has a deadlock}\right).$$

Here, "has a deadlock" means "there is a locally or globally deadlocked run".

For the systems and specifications presented in this work, cutoffs can be computed from the size of the process template B and the number k of copies of B mentioned in the specification, and are given as expressions like $|B| + k + 1$.

Remark 9. Our definition of a cutoff is different from that of Emerson and Kahlon [37], and instead similar to, e.g., Emerson and Namjoshi [40]. The reason is that we want the following property to hold for any (A, B) and Φ:

> if n_0 is the smallest number such that $\forall n \geq n_0 :\ (A,B)^{(1,n)} \models \Phi$, then any $c < n_0$ is not a cutoff, any $c \geq n_0$ is a cutoff.

We call n_0 the *tight* cutoff. The definition of Emerson and Kahlon [37, page 2] requires that $\forall n \leq c.(A,B)^{(1,n)} \models \Phi$ if and only if $\forall n \geq 1 : (A,B)^{(1,n)} \models \Phi$, and thus allows stating $c < n_0$ as a cutoff if Φ does not hold for all n.

[2]Here we do not explicitly mention local deadlocks because they can be specified as a part of $\mathsf{A}h(A, B^{(k)})$.

Parameterized synthesis. We encourage the reader to revisit Chapter 2 on page 24 to recall how bounded synthesis works in the case of non-distributed systems. Now we adapt the procedure to guarded parameterized systems. In parameterized model checking a cutoff allows us to check whether any "big" system satisfies the specification by checking it in the cutoff system. A similar reduction applies to the parameterized synthesis problem [50]. For guarded protocols, we obtain the following *semi-decision procedure for parameterized synthesis*:

0. set initial bound (b_A, b_B) on the size of the process templates;
1. determine the cutoff for (b_A, b_B) and Φ;
2. solve the bounded template synthesis problem for cutoff, size bound, and Φ;
3. if successful, return (A, B), else increase (b_A, b_B) and goto (1).

This procedure was implemented inside our parameterized synthesis tool PARTY [57] by Simon Außerlechner as a part of his Master Thesis [6].

Existing Cutoff Results

Emerson and Kahlon [37] have shown:

Theorem 13 (Disjunctive Cutoff Theorem)**.** *For closed disjunctive systems $A\|B^n$, $|B|+2$ is a cutoff $^{(\dagger)}$ for formulas of the form $\mathsf{A}h(A, B^{(1)})$ and $\mathsf{E}h(A, B^{(1)})$, and for global deadlock detection.*

Theorem 14 (Conjunctive Cutoff Theorem)**.** *For closed conjunctive systems $A\|B^n$, $2\,|B|$ is a cutoff $^{(\dagger)}$ for formulas of the form $\mathsf{A}h(A)$ and $\mathsf{E}h(A)$, and for global deadlock detection. For formulas of the form $\mathsf{A}h(B^{(1)})$ and $\mathsf{E}h(B^{(1)})$, $2\,|B|+1$ is a cutoff.*

In the above theorems, $h(A)$ (resp. $h(B^{(1)})$) means that the formula talks about the A-process only (resp. B_1).

Remark 10. (†) Note that Emerson and Kahlon [37] proved these results for a different definition of a cutoff (see Remark 9). Their results also hold for our definition, except possibly for global deadlocks. For the latter case to hold with the new cutoff definition, one also needs to prove the direction "global deadlock in the cutoff system implies global deadlock in a large system" (later called Monotonicity Lemma). In Sections 5.7.3 and 5.7.4, Sections 5.8.3 and 5.8.4, we prove these lemmas for the case of general deadlock (global *or* local).

Challenge: Open Systems

For any open system S there exists a closed system S' such that S and S' cannot be distinguished by LTL specifications (e.g., see Manna and Pnueli [65]). Thus, one approach to PMC for open systems is to use a translation between open and closed systems, and then use the existing cutoff results for closed systems.

While such an approach works in theory, it might not be feasible in practice: since cutoffs depend on the size of the process templates, and the translation blows up the process template, it also blows up the cutoffs. Thus, cutoffs that directly support open systems are important.

Challenge: Liveness and Deadlocks under Fairness

We are interested in cutoff results that support liveness properties. Consider a specification $\Phi = h(A, B^{(k)})$. In general, we would like to consider only runs where all processes move infinitely often, i.e., use the unconditional fairness assumption $\forall p.\, \mathsf{GF}\, \mathsf{move}_p$ and thus have $\mathsf{A}_{uncond}\Phi$. However, this would mean that we accept all systems that always go into a local deadlock, since then the assumption is violated (i.e., there will be no unconditionally-fair runs). This is especially undesirable in synthesis, because the synthesizer often tries to violate the assumptions to satisfy the specification. To avoid this, we require the absence of local deadlocks. But local deadlocks may appear due to unfair scheduling. Therefore we require the absence of local deadlocks under the strong fairness assumption, i.e., we require satisfaction of the formula $\mathsf{A}_{strong}\Phi_{\neg dead} = \big(\forall p.(\mathsf{GF}\, \mathsf{en}_p \rightarrow \mathsf{GF}\, \mathsf{move}_p)\big) \rightarrow \forall p.\, \mathsf{GF}\, \mathsf{en}_p$. This formula can be roughly read as "the absence of local deadlocks under fair scheduling". Since absence of global deadlocks and absence of local deadlocks under strong fairness imply unconditional fairness, we can safely use $\mathsf{A}_{uncond}\Phi$.

In summary, for a parameterized specification Φ, we consider satisfaction of

$$\textit{"all runs are infinite"} \quad \wedge \quad \mathsf{A}_{strong}\Phi_{\neg dead} \quad \wedge \quad \mathsf{A}_{uncond}\Phi.$$

This is equivalent to *"all runs are infinite"*$\wedge \mathsf{A}_{strong}(\Phi_{\neg dead} \wedge \Phi)$, but by considering the form above we can separate the tasks of deadlock detection and of model checking LTL\X-properties, and obtain modular cutoffs. (The phrase "all runs are infinite" is another way of saying "all runs have no global deadlocks".)

5.5 New Cutoff Results

We present new cutoff results that extend Theorems 13 and 14. The new and previous results are summarized in the table below.

	$h(A, B^{(k)})$ no fairness	deadlock detection no fairness	$h(A, B^{(k)})$ uncond. fairness	deadlock detection strong fairness
Disjunctive	$\lvert B\rvert + k + 1$	$2\lvert B\rvert - 1$	$2\lvert B\rvert + k - 1$	$2\lvert B\rvert - 1$
Conjunctive	$k + 1$	$2\lvert B\rvert - 2$ $(*)$	$k + 1$ $(*)$	$2\lvert B\rvert - 2$ $(*)$

The table distinguishes between disjunctive and conjunctive systems (in rows). In the columns, we consider satisfaction of properties $h(A, B^{(k)})$ and the existence of deadlocks, with and without fairness assumptions. All results hold for open systems, and for both path quantifiers A and E. Cutoffs depend on the size of process template B and the number $k \geq 1$ of B-processes a property talks about.

Results marked with a $(*)$ are for a restricted class of systems: for conjunctive systems with fairness, we require infinite runs to be initializing, i.e., all non-deadlocked processes return to init infinitely often.[3] Additionally, the cutoffs for deadlock detection in conjunctive systems only support 1-conjunctive systems.

All cutoffs in the table are tight—no smaller cutoff can exist for this class of systems and properties—except for the case of deadlock detection in disjunctive systems without fairness. There, the cutoff is asymptotically tight, i.e., it must increase linearly with the size of the process template.

Note that the table does not describe all possible combinations: for example, we do not consider satisfaction of $h(A, B^{(k)})$ on strong-fair runs. But the results in the table are the most interesting, from our view, for parameterized synthesis.

In the following sections we prove the results.

5.6 Proof Structure

The proofs for the cutoff results, new and original, are based on two lemmas, Monotonicity and Bounding [37]. When combined together, the lemmas give a cutoff. We state the lemmas, and discuss them in the context of deadlock detection and fairness. The detailed proofs are in Sections 5.7 and 5.8. Note that we only consider properties of the form $h(A, B^{(1)})$—the proof ideas extend to general properties $h(A, B^{(k)})$ without difficulty. Similarly, in most cases the proof ideas extend to open systems without major difficulties—mainly because when we construct a simulating run, we have the freedom to choose the input that is needed. Only for the case of deadlock detection we have to handle open systems explicitly.

1) Monotonicity lemma: if a behavior is possible in a (conjunctive or disjunctive) system with $n \in \mathbb{N}$ copies of B, then it is also possible in a (conjunctive or disjunctive resp.) system with one additional process:

$$(A, B)^{(1,n)} \models \mathsf{E}\, h(A, B^{(1)}) \implies (A, B)^{(1,n+1)} \models \mathsf{E}\, h(A, B^{(1)}),$$

and if a deadlock is possible in $(A, B)^{(1,n)}$, then it is possible in $(A, B)^{(1,n+1)}$.

Discussion. The lemma is easy to prove for properties $\mathsf{E}\, h(A, B^{(1)})$ in both disjunctive and conjunctive systems, by letting the additional process stay in its initial state init_B forever (see [37]). This cannot disable transitions with disjunctive guards, as these check for *existence* of a local state in another process (and we do not remove any processes), and it cannot disable conjunctive guards since they contain init_B by assumption. However, this construction violates fairness, since the new process never moves. This can be resolved in the disjunctive case by letting the additional process mimic all transitions of an existing process. But in general this does not work in conjunctive systems (due to the non-reflexive interpretation of guards). For this case and for deadlock detection, the proof is not trivial and may only work for $n \geq c$, for some lower bound $c \in \mathbb{N}$. The following sections provide the details.

[3] This assumption is in the same flavor as the restriction that init_A and init_B appear in all conjunctive guards. Intuitively, the additional restriction makes sense since conjunctive systems model shared resources, and everybody who takes a resource should eventually release it.

2) Bounding lemma: there exists a number $c \in \mathbb{N}$ such that a behavior is possible in a system with c copies of B if it is possible in a system with $n \geq c$ copies of process B:

$$(A, B)^{(1,c)} \models \mathsf{E}\, h(A, B^{(1)}) \iff (A, B)^{(1,n)} \models \mathsf{E}\, h(A, B^{(1)}),$$

and a deadlock is possible in $(A, B)^{(1,c)}$ if it is possible in $(A, B)^{(1,n)}$.

Discussion. For disjunctive systems, the main difficulty is that removing processes might falsify guards of the local transitions of A or B_1 in a given run. To address this, Emerson and Kahlon [37] came up with so-called flooding construction (described later). For conjunctive systems, removing processes from a run is easy for the case of infinite runs, since a transition that was enabled before cannot become disabled. Here, the difficulty is in preserving deadlocks, because removing processes may enable processes that were deadlocked before. The next sections explain how to address this.

Tightness. Recall from Section 5.4 that c is a tight cutoff iff c is a cutoff and there are templates (A, B) and a property Φ, such that

$$(A, B)^{(1,c-1)} \not\models \Phi \text{ and } (A, B)^{(1,c)} \models \Phi.$$

For deadlock detection this is equivalent to: $(A, B)^{(1,c-1)}$ does not have a deadlock but $(A, B)^{(1,c)}$ does. To prove tightness, we provide a template (A, B) and a property.

The next sections contains all the proofs of the results in the table. For each row and column, we prove monotonicity and bounding lemmas, as well as tightness. Note that for simplicity the proofs are for the case of $h(A, B_1)$, while the generalization to the case $h(A, B^{(k)})$ follows.

5.7 Proof Techniques for Disjunctive Systems

5.7.1 LTL\X Properties without Fairness: Existing Constructions

We revisit the main techniques of the original proof of Theorem 13 [37].

Lemma 15 (Monotonicity: Disj, LTL\X, Unfair)**.** *For disjunctive systems:*

$$\forall n \geq 1 :$$
$$(A, B)^{(1,n)} \models \mathsf{E}\, h(A, B_1) \;\Rightarrow\; (A, B)^{(1,n+1)} \models \mathsf{E}\, h(A, B_1).$$

Proof. Given a run x of $(A, B)^{(1,n)}$, we construct a run y of $(A, B)^{(1,n+1)}$: copy x into y and keep the additional process in the initial state. □

As for the bounding lemma, we construct an infinite run y of $(A, B)^{(1,c)}$ with $y \models h(A, B^{(1)})$, based on an infinite run x of $(A, B)^{(1,n)}$ with $n > c$ and $x \models h(A, B^{(1)})$. The idea is to copy local runs $x(A)$ and $x(B_1)$ into y, and construct runs of other processes in a way that enables all transitions along $x(A)$ and $x(B_1)$. The latter is achieved with the flooding construction.

Flooding construction [37]. Given a run $x = (s_1, e_1, p_1), (s_2, e_2, p_2) \dots$ of $(A, B)^{(1,n)}$, let $\mathsf{Visited}_{\mathcal{B}}(x)$ be the set of all local states visited by B-processes in x, i.e., $\mathsf{Visited}_{\mathcal{B}}(x) = \{q \in Q_B \mid \exists m \exists i.\ s_m(B_i) = q\}$.

For every $q \in \mathsf{Visited}_{\mathcal{B}}(x)$ there is a local run of $(A, B)^{(1,n)}$, say $x(B_i)$, that visits q first, say at moment m_q. Then, saying that process B_{i_q} of $(A, B)^{(1,c)}$ *floods* q means:

$$y(B_{i_q}) = x(B_i)[1\!:\!m_q](q)^\omega.$$

In words: the run $y(B_{i_q})$ is the same as $x(B_i)$ until moment m_q, and after that the process never moves.

The construction achieves the following. If we copy local runs of A and B_1 from x to y, and in y for every $q \in \mathsf{Visited}_{\mathcal{B}}(x)$ introduce one process that floods q, then: if in x at some moment m there is a process in state q', then in y at moment m there will also be a process (different from A and B_1) in state q'. Thus, every transition of A and B_1, which is enabled at moment m in x, will also be enabled in y.

Lemma 16 (Bounding: Disj, LTL\X, Unfair)**.** *For disjunctive systems:*

$$\forall n \geq |B| + 2 :\ (A, B)^{(1,|B|+2)} \models \mathsf{E}\, h(A, B_1) \quad \Leftarrow \quad (A, B)^{(1,n)} \models \mathsf{E}\, h(A, B_1).$$

The proof of the lemma is from [37, Lemma 4.1.2]. We recapitulate it to introduce the notions of "a process floods a state", $\mathsf{destutter}$, *interleave*, and "process mimics another process", which are used in our proofs later.

Proof idea. The lemma is proved by copying local runs $x(A)$ and $x(B_1)$, and flooding all states in $\mathsf{Visited}_{\mathcal{B}}(x)$. To ensure that at least one process moves infinitely often in y, we copy one additional (infinite) local run from x. Finally, it may happen that the resulting collection of local runs violates the interleaving semantics requirement. To resolve this, we add stuttering steps into local runs whenever two or more processes move at the same time, and we remove global stuttering steps in y. Since the only difference between $x(A, B_1)$ and $y(A, B_1)$ are stuttering steps, y and x satisfy the same LTL\X-properties $h(A, B^{(1)})$. Since $|\mathsf{Visited}_{\mathcal{B}}(x)| \leq |B|$, we need at most $1 + |B| + 1$ copies of B in $(A, B)^{(1,c)}$. □

Proof. Let $c = |B| + 2$ and $n \geq c$. Let $x = (s_1, e_1, p_1), (s_2, e_2, p_2) \dots$ be a run of $(A, B)^{(1,n)}$ that satisfies $\mathsf{E}\, h(A, B_1)$. We construct a run y of the cutoff system $(A, B)^{(1,c)}$ with $y(A, B_1) \simeq x(A, B_1)$.

Let $\mathsf{Visited}(x)$ be the set of all visited states by B-processes in run x: $\mathsf{Visited}(x) = \{q \mid \exists m \exists i : s_m(B_i) = q\}$.

Construct the run y of $(A, B)^{(1,c)}$ as follows.

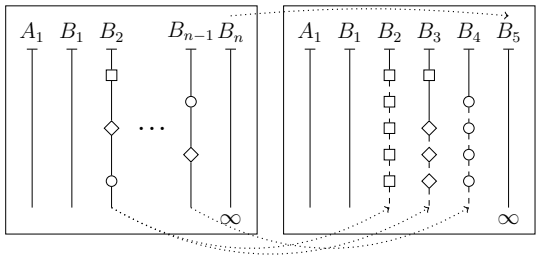
a. We copy runs of A and B_1 from x to y: $y(A) = x(A)$, $y(B_1) = x(B_1)$;

b. Since x is infinite, it has at least one infinitely moving process, denoted B_∞. Devote one unique process B_∞ in $(A, B)^{(1,c)}$ that copies the behaviour of B_∞ of $(A, B)^{(1,n)}$: $y(B_\infty) = x(B_\infty)$.

c. For every $q \in \mathsf{Visited}$, there is a process of $(A,B)^{(1,n)}$, denoted B_i, that visits q first, at moment denoted m_q. Then devote one unique process in $(A,B)^{(1,c)}$, denoted B_{i_q}, that *floods* q: set $y(B_{i_q}) = x(B_i)[1\!:\!m_q](q)^\omega$. In words: the run $y(B_{i_q})$ repeats exactly that of $x(B_i)$ till moment m_q, after which the process is never scheduled.

d. Let any other process B_i of $(A,B)^{(1,c)}$ not used in the previous steps (if any) *mimic* the behavior of B_1 of $(A,B)^{(1,c)}$: $y(B_i) = y(B_1)$.

The figure illustrates the construction. On the left is $(A,B)^{(1,n)}$ and on the right is $(A,B)^{(1,|B|+2)}$ (i.e., $(A,B)^{(1,5)}$, since $|B| = 3$ in the figure).



The correctness follows from the observation that any transition of any process at any moment m of y was done by some process in x at moment m, and hence is enabled at m. Also note that, if ≥ 2 processes transit simultaneously in y, then the guards of their transitions will be enabled even if both of them are removed from the state space. Note that it is possible that in y:

- more than one process transits at the same moment. Then, *interleave* the transitions of such processes, namely arbitrarily sequentialize them.
- at some moment no processes move. Then remove elements of the run y – the resulting run is denoted $\mathsf{destutter}(y)$.

This construction uses $|\mathsf{Visited}| + 2 \leq |B| + 2$ copies of B (ignoring case (d)). □

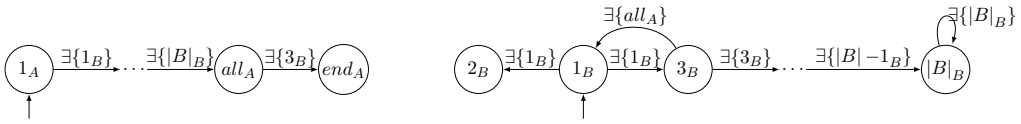
Tightness 1 (Disj, LTL\X, Unfair)**.** *The cutoff in Lemma 16 is tight. I.e., for any k there exist process templates (A,B) with $|B| = k$ and LTL\X formula $h(A,B_1)$ such that:*

$$(A,B)^{(1,|B|+2)} \models \mathsf{E}\, h(A,B_1) \quad \textit{and} \quad (A,B)^{(1,|B|+1)} \not\models \mathsf{E}\, h(A,B_1).$$

Proof. The idea of the proof relies on the subtleties of the definition of a run: it is infinite (thus not globally deadlocked), and in each step of a run exactly one process moves.

Consider the templates from Figure 5.1 and let $\mathsf{E}\, h(A,B_1) = \mathsf{E}(\mathsf{F}\, 3_{B_1} \wedge \mathsf{F}\, \mathsf{G}(2_{B_1} \wedge end_A))$. In words: there exists a run in a system where process B_1 visits 3_B and process B_1 with A eventually always stay in 2_B and end_A.

We need one process in every state of B to enable the transitions of A to all_A. Only when A in all_A, B_1 can move $3_B \to 1_B$, and then at some point to 2_B. After B_1 moves $3_B \to 1_B$, A moves $all_A \to end_A$, which requires process $B_{i\neq 1}$ in 3_B. Finally, to make the run infinite, there should be at least two processes in the state k_B. Hence, every infinite run satisfying the formula needs at least $|B| + 2$ B-processes. □



Template A Template B

Figure 5.1: Templates for proving Tightness 1

5.7.2 LTL\X Properties with Fairness: New Constructions

As for the case without fairness, proving the monotonicity lemma is simple.

Lemma 17 (Monotonicity: Disj, LTL\X, Fair)**.** *For disjunctive systems:*

$$\forall n \geq 1 :$$
$$(A, B)^{(1,n)} \models \mathsf{E}_{uncond}\, h(A, B_1) \implies (A, B)^{(1,n+1)} \models \mathsf{E}_{uncond}\, h(A, B_1),$$

Proof. In run x of $(A, B)^{(1,n)}$ with $n \geq 1$ all processes move infinitely often. Hence let the run y of $(A, B)^{(1,n+1)}$ copy x, and let the new process mimic an infinitely moving B process of $(A, B)^{(1,n)}$. □

To prove the bounding lemma, we introduce two new constructions. We need new constructions, because the flooding construction does not preserve fairness, and also cannot be used to construct deadlocked runs, since it does not preserve disabledness of transitions of processes A or B_1.

Consider the proof task of the bounding lemma for disjunctive systems with fairness: given an unconditionally fair run x of $(A, B)^{(1,n)}$ with $x \models h(A, B^{(1)})$, we want to construct an unconditionally fair run y of $(A, B)^{(1,c)}$ with $y \models h(A, B^{(1)})$. In contrast to unfair systems, we need to ensure that all processes move infinitely often in y. The insight is that after a finite time all processes will start looping around some set $\mathsf{Visited}^{inf}$ of states. We construct a run y that mimics this. To this end, we introduce two constructions. *Flooding with evacuation* is similar to flooding, but instead of keeping processes in their flooding states forever it evacuates the processes into $\mathsf{Visited}^{inf}$. *Fair extension* lets all processes move infinitely often without leaving $\mathsf{Visited}^{inf}$.

Flooding with evacuation. Given a subset $\mathcal{F} \subseteq \mathcal{B}$ and an infinite run $x = (s_1, e_1, p_1) \ldots$ of $(A, B)^{(1,n)}$, define

$$\mathsf{Visited}^{inf}_{\mathcal{F}}(x) = \{q \mid \exists \text{ infinitely many } \quad m\colon s_m(B_i) = q \text{ for some } B_i \in \mathcal{F}\} \quad (5.1)$$
$$\mathsf{Visited}^{fin}_{\mathcal{F}}(x) = \{q \mid \exists \text{ only finitely many } m\colon s_m(B_i) = q \text{ for some } B_i \in \mathcal{F}\} \quad (5.2)$$

Let $q \in \mathsf{Visited}^{fin}_{\mathcal{F}}(x)$. In run x there is a moment f_q when q is reached for the first time by some process from $\mathcal{F}$, denoted B_{first_q}. Also, in run x there is a moment l_q such that: $s_{l_q}(B_{\mathsf{last}_q}) = q$ for some process $B_{\mathsf{last}_q} \in \mathcal{F}$, and $s_t(B_i) \neq q$ for all $B_i \in \mathcal{F}$, $t > l_q$—i.e., when some process from $\mathcal{F}$ is in state q for the last time in x. Then, saying that process B_{i_q} of $(A, B)^{(1,c)}$ *floods* $q \in \mathsf{Visited}^{fin}_{\mathcal{F}}(x)$ *and then evacuates into* $\mathsf{Visited}^{inf}_{\mathcal{F}}(x)$ means:

$$y(B_{i_q}) = x(B_{\mathsf{first}_q})[1\colon f_q] \;\cdot\; (q)^{(l_q - f_q + 1)} \cdot\; x(B_{\mathsf{last}_q})[l_q\colon m] \;\cdot\; (q')^{\omega},$$

where q' is the state in $\mathsf{Visited}^{inf}_{\mathcal{F}}(x)$ that $x(B_{\mathsf{last}_q})$ reaches first, at some moment $m \geq l_q$. In words, process B_{i_q} mimics process B_{first_q} until it reaches q, then does nothing until process B_{last_q} starts leaving q, then it mimics B_{last_q} until it reaches $\mathsf{Visited}^{inf}_{\mathcal{F}}(x)$.

The construction ensures: if we copy local runs of all processes not in $\mathcal{F}$ from x to y, then all transitions of y are enabled. This is because, for any process p of $(A,B)^{(1,c)}$ that takes a transition in y at any moment, the set of states visible to process p is a superset of the set of states visible to the original process in $(A,B)^{(1,n)}$ whose transitions process p copies.

Fair extension. Here, we consider a path x that is the postfix of an unconditionally fair run x' of $(A,B)^{(1,n)}$, starting from the moment where no local states from $\mathsf{Visited}^{fin}_{\mathcal{B}}(x')$ are visited anymore. We construct a corresponding unconditionally-fair path y of $(A,B)^{(1,c)}$, where no local states from $\mathsf{Visited}^{fin}_{\mathcal{B}}(x')$ are visited.

Formally, let $n \geq 2|B|$, and x an unconditionally-fair path of $(A,B)^{(1,n)}$ such that $\mathsf{Visited}^{fin}_{\mathcal{B}}(x) = \emptyset$. Let $c \geq 2|B|$, and s'_1 a state of $(A,B)^{(1,c)}$ with

- $s'_1(A_1) = s_1(A_1)$, $s'_1(B_1) = s_1(B_1)$;
- for every $q \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \setminus \mathsf{Visited}^{inf}_{B_1}(x)$, there are two processes $B_{i_q}, B_{i'_q}$ of $(A,B)^{(1,c)}$ that start in q, i.e., $s'_1(B_{i_q}) = s'_1(B_{i'_q}) = q$;
- for every $q \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \cap \mathsf{Visited}^{inf}_{B_1}(x)$, there is one process B_{i_q} of $(A,B)^{(1,c)}$ that starts in q;
- for some $q^\star \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \cap \mathsf{Visited}^{inf}_{B_1}(x)$, there is one additional process of $(A,B)^{(1,c)}$, different from any in the above, called $B_{i'_{q^\star}}$, that starts in $q^\star$; and
- any other process B_i of $(A,B)^{(1,c)}$ starts in some state of $\mathsf{Visited}^{inf}_{B_2..B_n}(x)$.

Note that, if $\mathsf{Visited}^{inf}_{B_2..B_n}(x) \cap \mathsf{Visited}^{inf}_{B_1}(x) = \emptyset$, then the third and fourth prerequisite are trivially satisfied.

The fair extension extends state s'_1 of $(A,B)^{(1,c)}$ to an unconditionally-fair path $y = (s'_1, e'_1, p'_1)\ldots$ with $y(A_1,B_1) = x(A_1,B_1)$ as follows.

(a) $y(A_1) = x(A_1)$, $y(B_1) = x(B_1)$.

(b) For every $q \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \setminus \mathsf{Visited}^{inf}_{B_1}(x)$: in run x there is $B_i \in \{B_2..B_n\}$ that starts in q and visits it infinitely often. Let B_{i_q} and $B_{i'_q}$ of $(A,B)^{(1,c)}$ mimic B_i in turns: first B_{i_q} mimics B_i until it reaches q, then $B_{i'_q}$ mimics B_i until it reaches q, and so on.

(c) Arrange the states of $\mathsf{Visited}^{inf}_{B_2..B_n}(x) \cap \mathsf{Visited}^{inf}_{B_1}(x)$ in some order $(q^\star, q_1, \ldots, q_l)$. The processes $B_{i'_{q^\star}}, B_{i_{q^\star}}, B_{i_{q_1}}, \ldots, B_{i_{q_l}}$ behave as follows. Start with $B_{i'_{q^\star}}$: when B_1 enters $q^\star$ in y, it carries[4] $B_{i'_{q^\star}}$ from $q^\star$ to q_1, then carries $B_{i_{q_1}}$ from q_1 to q_2, …, then carries $B_{i_{q_l}}$ from q_l to $q^\star$, then carries $B_{i_{q^\star}}$ from $q^\star$ to q_1, then carries $B_{i'_{q^\star}}$ from q_1 to q_2, then carries $B_{i_{q_1}}$ from q_2 to q_3, and so on.

[4] "Process B_1 starting at moment m carries process B_i from q to q'" means: process B_i mimics the transitions of B_1 starting at moment m at q until B_1 first reaches q'.

(d) Any other B_i of $(A,B)^{(1,c)}$, starting in $q \in \mathsf{Visited}^{inf}_{B_2..B_n}(x)$, mimics B_{i_q}.

Note that parts (b) and (c) of the construction ensure that there is always at least one process in every state from $\mathsf{Visited}^{inf}_{B_2..B_n}(x)$. This ensures that the guards of all transitions of the construction are satisfied. Excluding processes in (d), the fair extension uses up to $2|B|$ copies of B.[5]

Now we are ready to prove the bounding lemma.

Lemma 18 (Bounding: Disj, LTL\X, Fair)**.** *For disjunctive systems:*

$$\forall n > 2|B| :$$
$$(A,B)^{(1,2|B|)} \models \mathsf{E}_{uncond}\, h(A,B_1) \qquad \Longleftarrow \qquad (A,B)^{(1,n)} \models \mathsf{E}_{uncond}\, h(A,B_1),$$

Proof. Let $c = 2\,|B|$. Given an unconditionally-fair run x of $(A,B)^{(1,n)}$, we construct an unconditionally-fair run y of the cutoff system $(A,B)^{(1,c)}$ such that $y(A,B_1)$ is stuttering equivalent to $x(A,B_1)$.

Note that in x there is a moment m such that all local states that are visited after m are in $\mathsf{Visited}^{inf}_{\mathcal{B}}(x)$.

The construction has two phases. In the first phase, we apply flooding for states in $\mathsf{Visited}^{inf}_{\mathcal{B}}(x)$, and flooding with evacuation for states in $\mathsf{Visited}^{fin}_{\mathcal{B}}(x)$:

(a) $y(A) = x(A)$, $y(B_1) = x(B_1)$;

(b) for every $q \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \backslash \mathsf{Visited}^{inf}_{B_1}(x)$, devote two processes of $(A,B)^{(1,c)}$ that flood q;

(c) for some $q^\star \in \mathsf{Visited}^{inf}_{B_2..B_n}(x) \cap \mathsf{Visited}^{inf}_{B_1}(x)$, devote one process of $(A,B)^{(1,c)}$ that floods $q^\star$;

(d) for every $q \in \mathsf{Visited}^{fin}_{B_2..B_n}(x)$, devote one process of $(A,B)^{(1,c)}$ that floods q and evacuates into $\mathsf{Visited}^{inf}_{B_2..B_n}(x)$; and

(e) let other processes (if any) mimic process B_1.

The phase ensures that at moment m in y, there are no processes in $\mathsf{Visited}^{fin}_{\mathcal{B}}(x)$, and all the pre-requisites of the fair extension are satisfied.

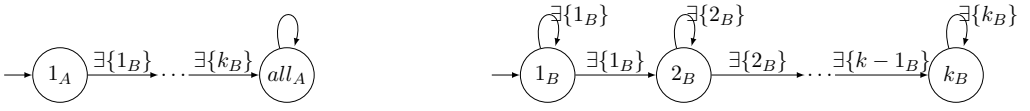
The second phase applies the fair extension, and then establishes the interleaving semantics as in the bounding lemma in the non-fair case. The overall construction uses up to $2|B|$ copies of B. □

Tightness 2 (Disj, LTL\X, Fair)**.** *The cutoff in Lemma 18 is tight. I.e., for any k there exist process templates (A,B) with $|B| = k$ and LTL\X formula $h(A,B_1)$ such that:*

$$(A,B)^{(1,2|B|)} \models \mathsf{E}\, h(A,B_1) \;\; \textit{and} \;\; (A,B)^{(1,2|B|-1)} \not\models \mathsf{E}\, h(A,B_1).$$

Proof. Consider process templates A, B from Figure 5.2 and the property $\mathsf{E}\,\mathsf{true}$. □

[5] A careful reader may notice that, if $|\mathsf{Visited}^{inf}_{B_1}(x)| = 1$ and $|\mathsf{Visited}^{inf}_{B_2..B_n}(x)| = |B|$, then the construction uses $2|B|+1$ copies of B. But one can slightly modify the construction for this special case, and remove process $B_{i'_{q^\star}}$ from the pre-requisites.



Template A Template B

Figure 5.2: Templates for proving Tightness 2

5.7.3 Deadlocks without Fairness: Updated Constructions

Lemma 19 (Monotonicity: Disj, Deadlocks, Unfair)**.** *For disjunctive systems:*

$$\forall n \geq |B| + 1 : (A, B)^{(1,n)} \text{ has a deadlock } \Rightarrow \ (A, B)^{(1,n+1)} \text{ has a deadlock.}$$

Proof. Given a deadlocked run x of $(A, B)^{(1,n)}$, we build a deadlocked run of $(A, B)^{(1,n+1)}$. If the run x is locally deadlocked, then it has at least one infinitely moving process, thus let the additional process mimic that process. If the run x is globally deadlocked run, then due to $n > |B|$ in some state there are at least two processes deadlocked. Thus, let the new process mimic a process deadlocked in that state—the run constructed will also be globally deadlocked. □

Lemma 20 (Bounding: Disj, Deadlocks, Unfair)**.** *For disjunctive systems:*

- *with* $c = |B| + 2$ *and any* $n > c$*:*

$$(A, B)^{(1,c)} \text{ has a local deadlock } \Leftarrow \ (A, B)^{(1,n)} \text{ has a local deadlock;}$$

- *with* $c = 2|B| - 1$ *and any* $n > c$

$$(A, B)^{(1,c)} \text{ has a global deadlock } \Leftarrow \ (A, B)^{(1,n)} \text{ has a global deadlock;}$$

- *with* $c = 2|B| - 1$ *and any* $n > c$*:*

$$(A, B)^{(1,c)} \text{ has a deadlock } \Leftarrow \ (A, B)^{(1,n)} \text{ has a deadlock.}$$

Proof idea. First, consider the case of global deadlocks. The insight is to divide deadlocked local states into two disjoint sets, dead_1 and dead_2, as follows. Given a globally deadlocked run x of $(A, B)^{(1,n)}$, for every $q \in \mathsf{dead}_1$, there is a process of $(A, B)^{(1,n)}$ deadlocked in q with input i, that has an outgoing transition guarded "$\exists q$"—hence, adding one more process into q would unlock the process. In contrast, $q \in \mathsf{dead}_2$ if any process deadlocked in q stays deadlocked after adding more processes into q. Let us denote the set of B-processes deadlocked in dead_1 by $\mathcal{D}_1$. Finally, abuse the definition in Eq. 5.2 and denote by $\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$ the set of states that are visited by B-processes not in $\mathcal{D}_1$ before reaching a deadlocked state.

Given a globally deadlocked run x of $(A, B)^{(1,n)}$ with $n \geq 2|B|-1$, we construct a globally deadlocked run y of $(A, B)^{(1,c)}$ with $c = 2|B| - 1$ as follows.

- We copy from x into y the local runs of processes in $\mathcal{D}_1 \cup \{A\}$;
- flood every state of dead_2; and

- for every $q \in \mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$, flood q and evacuate into dead_2.

The construction ensures: (1) for any moment and any process in y, the set of local states that are visible to the process includes all the states that were visible to the corresponding process in $(A,B)^{(1,n)}$ whose transitions we copy; (2) in y, there is a moment when all processes deadlock in $\mathsf{dead}_1 \cup \mathsf{dead}_2$.

For the case of local deadlocks, the construction is slightly more involved, since we also need to copy the behaviour of an infinitely moving process. □

Proof. Given a (globally or locally) deadlocked run of $(A,B)^{(1,n)}$, we construct (globally or locally) deadlocked run of $(A,B)^{(1,c)}$, where c depends on the nature of the given run. We do this using the construction template.

Let $\mathcal{B} = \{B_1, ..., B_n\}$. The template depends on the set $\mathcal{C} \subseteq \{B_1, ..., B_c\}$ and is as follows.

a. Set $y(A) = x(A)$;

b. for every $B_i \in \mathcal{C}$, set $y(B_i) = x(B_i)$;

c. for every $q \in \mathsf{Visited}^{inf}_{\mathcal{B}\mathcal{C}}(x)$, devote one process of $(A,B)^{(1,c)}$ that floods q;

d. for every $q \in \mathsf{Visited}^{fin}_{\mathcal{B}\mathcal{C}}(x)$, devote one process of $(A,B)^{(1,c)}$ that floods q and then evacuates into $\mathsf{Visited}^{inf}_{\mathcal{B}\mathcal{C}}(x)$; and

e. let other processes (if any) mimic some process from (c).

1) Local deadlock. We distinguish three cases:

1a) A deadlocks, B_1 moves infinitely often;

1b) A moves infinitely often, B_1 deadlocks; and

1c) A neither deadlocks nor moves infinitely often, B_1 deadlocks, B_2 moves infinitely often.

1a: "A deadlocks, B_1 moves infinitely often".

Let $c = |B| + 1$, and $\mathcal{C} = \{B_1\}$. Note that $\mathsf{Visited}^{inf}_{B_2..B_n}(x) \neq \emptyset$. The resulting construction uses $|\mathsf{Visited}^{fin}_{B_2..B_n}(x)| + |\mathsf{Visited}^{inf}_{B_2..B_n}(x)| + 1 \leq |B| + 1$ copies of B.

1b: "A moves infinitely often, B_1 deadlocks".

Let $c = |B| + 1$, and $\mathcal{C} = \{B_1\}$. Let $q_\perp$ be the state in which B_1 deadlocks. Instantiate the construction template.

Process B_1 of $(A,B)^{(1,c)}$ is deadlocked in y starting from some moment d, because any state it sees (in $\mathsf{Visited}^{inf}_{A,B_2..B_n}(x)$) was also seen by B_1 in $(A,B)^{(1,n)}$ in x at some moment $d' \geq d$ (note that d' may be not the same moment as d).

1c: "A neither deadlocks nor moves infinitely often, B_1 deadlocks, B_2 moves infinitely often".

Instantiate the construction template with $c = |B| + 2$ and $\mathcal{C} = \{B_1, B_2\}$.

Finally, $|B| + 2$ is a (possibly not tight) cutoff for local deadlock detection problem.

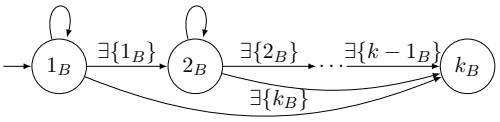


Figure 5.3: Templates for proving Tightness 3

2) Global deadlock. Let $x = (s_1, e_1, p_1)...(s_d, e_d, \perp)$ be a globally deadlocked run of $(A, B)^{(1,n)}$ with $n \geq c$.

Let us abuse the definition of $\mathsf{Visited}^{inf}_{\mathcal{F}}(x)$ and $\mathsf{Visited}^{fin}_{\mathcal{F}}(x)$, in Eq. 5.1 and 5.2 resp., and adapt it to the case of finite runs. To this end, given a finite run $x = (s_1, e_1, p_1)...(s_d, e_d, \perp)$, extend it to the infinite sequence $(s_1, e_1, p_1)...(s_d, e_d, \perp)^{\omega}$, and apply the definition of $\mathsf{Visited}^{inf}_{\mathcal{F}}(x)$ and $\mathsf{Visited}^{fin}_{\mathcal{F}}(x)$ to the sequence.

Let $\mathcal{D}_1$ be the set of processes deadlocked in unique states: $\forall p \in \mathcal{D}_1 \nexists p' \neq p : s_d(p') = s_d(p)$. Instantiate the construction template with $\mathcal{C} = \mathcal{D}_1$ and $c = 2|B|-1$. [6]

3) Deadlocks. As the cutoff for the deadlock detection problem we take the largest cutoff in (1)–(2), namely, $2|B| - 1$, but it may be not tight—finding the tight cutoffs for local deadlock and for deadlock detection problems is an open problem.

□

Tightness 3 (Disj, Deadlocks, Unfair)**.** *The cutoff $c = 2|B| - 1$ for deadlock detection in disjunctive systems is* asymptotically optimal but possibly not tight*. I.e., for any k there are templates (A, B) with $|B| = k$ such that:*

$$(A, B)^{(1,|B|-1)} \textit{ does not have a deadlock, but } (A, B)^{(1,|B|)} \textit{ does.}$$

Proof. Figure 5.3 illustrates templates (A, B) to prove the asymptotic optimality of cutoff $2|B| - 1$ for deadlock detection problem. Template A is any that never deadlocks. The system has a local deadlock only when there are at least $|B|$ copies of B, which is a constant factor of $2|B| - 1$. □

5.7.4 Deadlocks with Fairness: New Constructions

Lemma 21 (Monotonicity: Disj, Deadlocks, Fair)**.** *For disjunctive systems, on strong-fair or finite runs:*

$$\forall n \geq |B| + 1 : (A, B)^{(1,n)} \textit{ has a deadlock } \Rightarrow\ (A, B)^{(1,n+1)} \textit{ has a deadlock.}$$

Proof. See proof of Lemma 19. □

Lemma 22 (Bounding: Disj, Deadlocks, Fair)**.** *For disjunctive systems, on strong-fair or finite runs:*

[6] $2|B| - 1$ copies is enough, because: $\mathsf{Visited}^{fin}_{\mathcal{BC}}(x) \cap \mathsf{Visited}^{inf}_{\mathcal{BC}}(x) = \emptyset$, $\mathsf{Visited}^{inf}_{\mathcal{BC}}(x) \cap \mathsf{Visited}^{inf}_{\mathcal{C}}(x) = \emptyset$, and if $\mathsf{Visited}^{fin}_{\mathcal{BC}}(x) \neq \emptyset$, then $\mathsf{Visited}^{inf}_{\mathcal{BC}}(x) \neq \emptyset$.

- *with* $c = 2|B| - 1$ *and any* $n > c$*:*

$$(A, B)^{(1,c)} \text{ has a local deadlock } \Leftarrow (A, B)^{(1,n)} \text{ has a local deadlock;}$$

- *with* $c = 2|B| - 1$ *and any* $n > c$

$$(A, B)^{(1,c)} \text{ has a global deadlock } \Leftarrow (A, B)^{(1,n)} \text{ has a global deadlock;}$$

- *with* $c = 2|B| - 1$ *and any* $n > c$*:*

$$(A, B)^{(1,c)} \text{ has a deadlock } \Leftarrow (A, B)^{(1,n)} \text{ has a deadlock.}$$

The proofs are similar to that of Lemma 20 (the case without fairness): the case of global deadlocks is exactly the same, the case of local deadlocks differ—we additionally use the fair extension to ensure the resulting run is fair.

Proof. If $(A, B)^{(1,n)}$ has a global deadlock, then the fairness does not influence the cutoff, and the proof from Lemma 20, case "Global Deadlocks", applies and gives the cutoff $2|B| - 1$. Hence below consider only the case of local deadlocks.

Given a strong-fair deadlocked run x of $(A, B)^{(1,n)}$, we first construct a strong-fair deadlocked run y of $(A, B)^{(1,c)}$ with $c = 2|B|$ and then argue that c can be reduced to $2|B| - 1$. The construction is similar to that in Lemma 20 – the differences originate from the need to infinitely move non deadlocked processes.

Let $\mathsf{dead}_{<2}(x)$ be the set of deadlocked states in the run x that are only deadlocked if there is no other process in the same state, and let $\mathcal{D}_1$ be the set of processes deadlocked in the run x in $\mathsf{dead}_{<2}(x)$. Let $\mathsf{dead}_2(x)$ be the set of states that are deadlocked in the run x even if there is another process in the same state.

We note the following:

- $|\mathcal{D}_1| = |\mathsf{dead}_{<2}(x)| \leq |B|$;
- $\mathsf{dead}_{<2}(x) \cap \mathsf{dead}_2(x) = \emptyset$;
- $\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \cap \mathsf{dead}_{<2}(x) \neq \emptyset$ is possible, because a state from $\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$ can first be visited by a process in $\mathcal{B}\setminus\mathcal{D}_1$, and later be deadlocked because of the process in $\mathcal{D}_1$;
- $\mathsf{dead}_2(x) \subseteq \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}^1}(x)$, and hence $\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \cap \mathsf{dead}_2(x) = \emptyset$.

The construction has two phases. The first phase is as follows.

a. For every $p \in \{A\} \cup \mathcal{D}_1$, set $y(p) = x(p)$;

b. for every $q \in \mathsf{dead}_2(x)$, devote one process of $(A, B)^{(1,c)}$ that floods it;

c. for every $q \in \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)$, devote two processes of $(A, B)^{(1,c)}$ that flood it;

d. for every $q \in \mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$, devote one process of $(A, B)^{(1,c)}$ that floods it and then evacuates into $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$; and

e. let other processes (if any) mimic some process from (c).

After this phase all B processes will be in $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \cup \mathsf{dead}_{<2}(x)$.

The second phase applies to processes in $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)$ the fair extension[7].

How many processes does the construction use? Note that the sets $\mathsf{dead}_{<2}(x) \cup \mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$, $\mathsf{dead}_2(x)$, $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)$ are disjoint, thus:

$$|\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)| + |\mathsf{dead}_{<2}(x)| + |\mathsf{dead}_2(x)| + 2|\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)| \leq \quad (5.3)$$

$$2|\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \cup \mathsf{dead}_{<2}(x)| + |\mathsf{dead}_2(x)| + 2|\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)| \leq \quad (5.4)$$

$$|B| + |\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \cup \mathsf{dead}_{<2}(x)| + |\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)| \leq 2|B|$$

Let us reduce the estimate to $\leq 2|B| - 1$:

- assume that $\mathsf{dead}_2(x) = \emptyset$ (otherwise, Eq.5.3 and the sets disjointness give $2|B| - 1$); and
- assume that $\mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x) \neq \emptyset$ (the other case together with eq.5.4, the sets disjointness, and the first item gives $2|B| - 1$);
- hence, the construction in step (d) evacuates the process in $q \in \mathsf{Visited}^{fin}_{\mathcal{B}\setminus\mathcal{D}_1}(x)$ into $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}_1}(x)\setminus\mathsf{dead}_2(x)$. Hence modify step (c) of the construction and for q devote a single process of $(A, B)^{(1,c)}$ that floods it. This will give $\leq 2|B|-1$.

This concludes the proof.

□

Tightness 4 (Disj, Deadlocks, Fair)**.** *The cutoff $c = 2|B|-1$ for deadlock detection in disjunctive systems on strong-fair or finite runs is tight. I.e., for any k there are templates (A, B) with $|B| = k$ such that:*

$$(A, B)^{(1,2|B|-2)} \text{ does not have a deadlock, but } (A, B)^{(1,2|B|-1)} \text{ does.}$$

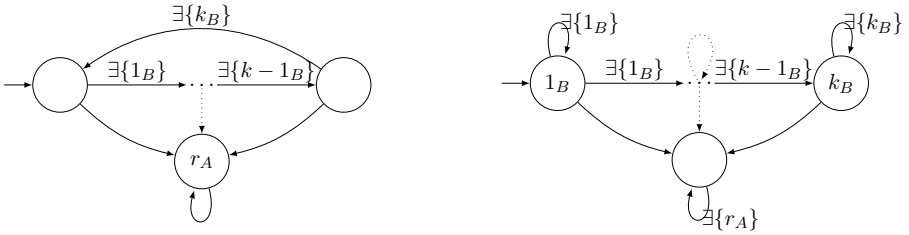
Proof. Figure 5.4 shows process templates (A, B) such that any system $(A, B)^{(1,n)}$ with $n \leq 2|B| - 2$ does not deadlock on strong-fair runs, but larger systems do. □

5.8 Proof Techniques for Conjunctive Systems

5.8.1 LTL\X Properties Without Fairness: Existing Constructions

The Monotonicity Lemma is proven [37] by keeping the additional process in the initial state.

[7]The fair extension requires the run x to be unconditionally-fair, but here we have a run in which all processes that are not deadlocked move infinitely often. To adapt the construction to this case: copy local runs of processes $\{A\} \cup \mathcal{D}_1$, and do not extend local runs of processes that are in a state in dead_2.



Template A Template B

Figure 5.4: Templates (A, B) used in Tightness 4.

Lemma 23 (Monotonicity: Conj, LTL\X, Unfair)**.** *For conjunctive systems,*

$$\forall n \geq 1 : \ (A,B)^{(1,n)} \models \mathsf{E}\, h(A, B_1) \quad \Rightarrow \quad (A,B)^{(1,n+1)} \models \mathsf{E}\, h(A, B_1).$$

Proof. Let the new process stutter in init state. □

To prove the Bounding Lemma, Emerson and Kahlon [37] suggest to simply copy the local runs $x(A)$ and $x(B_1)$ into y. In addition, we may need one more process that moves infinitely often to ensure that an infinite run of $(A, B)^{(1,n)}$ will result in an infinite run of $(A, B)^{(1,c)}$. All transitions of copied processes will be enabled because removing processes from a conjunctive system cannot disable a transition that was enabled before.

Lemma 24 (Bounding: Conj, LTL\X, Unfair)**.** *For conjunctive systems,*

$$\forall n \geq 2 : \ (A,B)^{(1,2)} \models \mathsf{E}\, h(A, B_1) \quad \Leftarrow \quad (A,B)^{(1,n)} \models \mathsf{E}\, h(A, B_1).$$

The proof is inspired by the first part of the proof of [37, Lemma 5.2].

Proof. Let $x = (s_1, e_1, p_1)(s_2, e_2, p_2)\ldots$ be a run of $(A, B)^{(1,n)}$. Note that, by the semantics of conjunctive guards, the transitions along any local run of x will also be enabled in any system $(A, B)^{(1,c)}$ with $c \leq n$, where the processes exhibit a subset of the local runs of x. Thus, we obtain a run of $(A, B)^{(1,c)}$ by copying a subset of the local runs of x, and removing elements of the new global run where all processes stutter.

Then, based on an infinite run x of the original system, we construct an infinite run y of the cutoff system. Let $y(A) = x(A)$ and $y(B_1) = x(B_1)$. The second copy of template B in $(A, B)^{(1,2)}$ is needed to ensure that the run y is infinite, i.e., at least one process moves infinitely often. If both $x(A)$ and $x(B_1)$ eventually deadlock, then there exists a process B_i of $(A, B)^{(1,n)}$ that makes infinitely many moves, and we set $y(B_2) = x(B_i)$. Otherwise, we set $y(B_2) = x(B_2)$. □

Tightness 5 (Conj, LTL\X, Unfair)**.** *The cutoff $c = 2$ is tight for parameterized model checking of properties $\mathsf{E}\, h(A, B_1)$ in the 1-conjunctive systems, i.e., there is a system type (A, B) and property $Eh(A, B_1)$ which is not satisfied by $(A, B)^{(1,1)}$ but is by $(A, B)^{(1,2)}$.*

Proof. Figure 5.5 shows templates (A, B), $\mathsf{E}\, h(A, B_1) = \mathsf{E}\,\mathsf{F}\, b$. An infinite run that satisfies the formula needs one copy of B that stays in the initial state, and one that moves into b. □

Template A Template B

Figure 5.5: Templates used to prove Tightness 5

5.8.2 LTL\X Properties with Fairness: New Constructions

In this section, subscript i in path quantifiers, E_i and A_i, denotes the quantification over initializing runs.

The proof of the Bounding Lemma is the same as in the non-fair case, noting that, if the original run is unconditional-fair, then so will be the resulting run.

Lemma 25 (Bounding: Conj, LTL\X, Fair)**.** *For unconditionally-fair initializing runs of conjunctive systems:*

$$\forall n \geq 1 :$$
$$(A,B)^{(1,1)} \models \mathsf{E}_{uncond}\, h(A,B_1) \;\Leftarrow\; (A,B)^{(1,n)} \models \mathsf{E}_{uncond}\, h(A,B_1).$$

Proof. Given an unconditionally-fair [initializing] run x of $(A,B)^{(1,n)}$ with $n > c$ construct an unconditionally-fair [initializing] run y in the cutoff system $(A,B)^{(1,1)}$: copy the local runs of processes A, B_1. □

Proving the Monotonicity Lemma is more difficult, since the fair extension construction from disjunctive systems does not work for conjunctive systems—if an additional process mimics the transitions of an existing process then it disables transitions of the form $q \overset{\text{“}\forall\neg q\text{”}}{\rightarrow} q'$ or $q \overset{\text{“}\forall\neg q'\text{”}}{\rightarrow} q'$. Hence, we add the restriction of initializing runs, which allows us to construct a fair run as follows. The additional process B_{n+1} “shares” a local run $x(B_i)$ with an existing process B_i of $(A,B)^{(1,n+1)}$: one process stutters in init_B while the other makes transitions from $x(B_i)$, and whenever $x(B_i)$ enters init_B (this happens infinitely often), the roles are reversed. Since this changes the behavior of B_i, B_i should not be mentioned in the formula, i.e., we need $n \geq 2$ for a formula $h(A, B^{(1)})$.

Lemma 26 (Monotonicity: Conj, LTL\X, Fair)**.** *For unconditionally-fair initializing runs of conjunctive systems:*

$$\forall n \geq 2 :$$
$$(A,B)^{(1,n)} \models \mathsf{E}_{uncond,i}\, h(A,B_1) \;\Rightarrow\; (A,B)^{(1,n+1)} \models \mathsf{E}_{uncond,i}\, h(A,B_1).$$

Proof. Given a unconditionally-fair initializing run x of $(A,B)^{(1,n)}$, we construct a unconditionally-fair initializing run y in $(A,B)^{(1,n+1)}$, with one additional process p. First, copy all local runs of all processes of $(A,B)^{(1,n)}$ from the run x into y. Then, let process p' stutter in init until some other process $p \neq B_1$ enters init. Then, exchange the roles of processes p' and p: let p stutter in init, while p' takes the transitions of p from the original run, until it enters init. And so on. In this way, we continue to interleave the run between p' and p, and obtain a unconditionally-fair initializing run for all processes, with $y(A,B_1) = x(A,B_1)$. Thus, if $(A,B)^{(1,n)} \models \mathsf{E}\, h(A,B_1)$, then $(A,B)^{(1,n+1)} \models \mathsf{E}\, h(A,B_1)$. □

Template A Template B

Figure 5.6: Templates used to prove Tightness 6

Tightness 6 (1-Conj, LTL\X, Fair)**.** *The cutoff $c = 2$ is tight for parameterized model checking of $\mathsf{E}\, h(A, B_1)$ on unconditionally-fair initializing runs in 1-conjunctive systems, i.e., there is a system type (A, B) and property $\mathsf{E}\, h(A, B_1)$ which is satisfied by $(A, B)^{(1,1)}$ but not by $(A, B)^{(1,2)}$.*

Proof. Figure 5.6 shows templates (A, B); $\mathsf{E}\, h(A, B_1) = \mathsf{E}\,\mathsf{FG}(b_{init} \rightarrow a_1)$. □

5.8.3 Deadlocks Without Fairness: Updated Constructions

Lemma 27 (Monotonicity: Conj, Deadlocks, Unfair)**.** *For conjunctive systems:*

$$\forall n \geq 1 : (A, B)^{(1,n)} \text{ has a deadlock } \Rightarrow\ (A, B)^{(1,n+1)} \text{ has a deadlock.}$$

Proof. Given a deadlocked run x of $(A, B)^{(1,n)}$, we construct a deadlocked run of $(A, B)^{(1,n+1)}$. Let y copy run x, and keep the new process in init. If x is globally deadlocked and d is the moment when the deadlock happens in x, then schedule the new process arbitrarily after moment d. Thus, it is possible that the newly constructed system run is only locally deadlocked, while the original run is globally deadlocked. □

As for the Bounding Lemma, in the case of global deadlock detection, Emerson and Kahlon [37] suggest to copy a subset of the original local runs. For every local state q that is present in the final state of the run, we need at most two local runs that end in this state. In the case of local deadlocks, our construction uses the fact that systems are 1-conjunctive. In 1-conjunctive systems, if a process is deadlocked, then there is a set of states *DeadGuards* that all need to be populated by other processes in order to disable all transitions of the deadlocked process. Thus, the construction copies: (i) the local run of a deadlocked process, (ii) for each $q \in \mathit{DeadGuards}$, the local run of a process that is in q at the moment of the deadlock, and (iii) the local run of an infinitely moving process.

Lemma 28 (Bounding: 1-Conj, Deadlocks, Unfair)**.** *For 1-conjunctive systems:*

- *with $c = 2|Q_B \setminus \{\mathsf{init}\}|$ and any $n > c$* [8]

$$(A, B)^{(1,c)} \text{ has a global deadlock } \Leftarrow\ (A, B)^{(1,n)} \text{ has a global deadlock;}$$

- *with $c = |Q_B \setminus \{\mathsf{init}\}| + 2$ and any $n > c$:*

$$(A, B)^{(1,c)} \text{ has a local deadlock } \Leftarrow\ (A, B)^{(1,n)} \text{ has a local deadlock;}$$

[8] This statement also applies to systems without restriction to 1-conjunctive guards.

- *with* $c = 2|Q_B \setminus \{\mathsf{init}\}|$ *and any* $n > c$*:*

$$(A,B)^{(1,c)} \text{ has a deadlock } \Leftarrow (A,B)^{(1,n)} \text{ has a deadlock.}$$

Proof. The proof is inspired by the second part of the proof of [37, Lemma 5.2], but in addition to global we consider local deadlocks.

Global deadlocks. Let $c = 2|Q_B \setminus \{\mathsf{init}\}|)$. Let run $x = (s_1, e_1, p_1) \dots (s_d, e_d, \perp)$ of $(A,B)^{(1,n)}$ with $n > c$ be globally deadlocked. We construct a globally deadlocked run y in $(A,B)^{(1,c)}$ as follows.

a. For every $q \in s_d \setminus \{\mathsf{init}\}$:
   - if s_d has two processes in state q, then devote two processes of $(A,B)^{(1,c)}$ that mimic the behaviour of the two of $(A,B)^{(1,n)}$ correspondingly;
   - otherwise, s_d has only one process in state q, then devote one process of $(A,B)^{(1,c)}$ that mimics the process of $(A,B)^{(1,n)}$;

b. for every process of $(A,B)^{(1,c)}$ not used in the construction (if any): let it mimic an arbitrary B-process of $(A,B)^{(1,n)}$ that was not yet used in the construction in item (a) nor (b).

The construction uses $\leq 2|Q_B \setminus \{\mathsf{init}\}|$ processes B. Note that the proof does not assume that the system is 1-conjunctive.

Local deadlocks. Let $c = |Q_B \setminus \{\mathsf{init}\}| + 2$. Let run $x = (s_1, e_1, p_1) \dots$ of $(A,B)^{(1,n)}$ with $n > c$ be locally deadlocked. We will construct a run y of $(A,B)^{(1,c)}$ where at least one process deadlocks and exactly one process moves infinitely often.

Wlog. we distinguish three cases:

1. A moves infinitely often in x, and B_1 deadlocks;
2. A deadlocks, and B_1 moves infinitely often; and
3. A neither deadlocks nor moves infinitely often, B_1 deadlocks, B_2 moves infinitely often.

1. "A moves infinitely often in x, and B_1 deadlocks".

Let $q_\perp, e_\perp$ be the deadlocked state and input of B_1 in x, and let d be the moment from which B_1 is deadlocked.

Let $\mathit{DeadGuards} = \{q_1, \dots, q_k\}$ be the set of states such that for every $q_i \in \mathit{DeadGuards}$ there is an outgoing transitions from $q_\perp$ with $e_\perp$ guarded "$\forall \neg q_i$", and assume $\mathit{DeadGuards} \neq \emptyset$ (if it is empty, then we keep every process in init until someone reaches $q_\perp$ and then schedule the rest arbitrarily). (Recall that $q_i \in Q_B \dot\cup Q_A$.)

The construction is as follows.

a. $y(A) = x(A)$, $y(B_1) = x(B_1)$.

b. For each $q \in \mathit{DeadGuards}$, at moment d in x there is a process p_q in state q. If $p_q \in \{B_1, ..., B_n\}$, then let one process of $(A,B)^{(1,c)}$ mimic it till moment d, and then stutter in q.

c. Let other processes of $(A, B)^{(1,c)}$ (if any) stay in init.

The construction uses (if ignore (c)) $\leq |Q_B \setminus \{\mathsf{init}\}| + 1$ processes B.

Note: the assumption of 1-conjunctive systems implies that, in order to deadlock B_1, we need a process in each state in *BlockGuards*. This implies that having a process in each state of *BlockGuards* does not disable any A's transition after moment d.

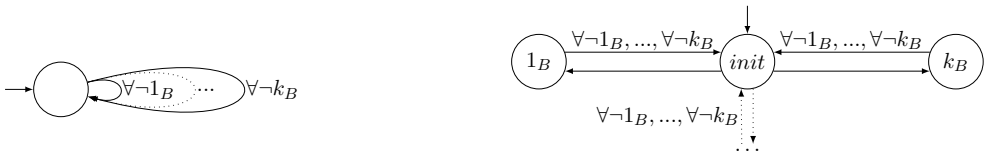
2. "A deadlocks, and B_1 moves infinitely often": use the construction from (1).

3. "A neither deadlocks nor moves infinitely often, B_1 deadlocks, B_2 moves infinitely often". Use the construction from (1), and additionally: $y(B_2) = x(B_2)$. Thus, the construction uses (if ignore (c)) $\leq |Q_B \setminus \{\mathsf{init}\}| + 2$ processes B.

Deadlocks. Take the higher value among the cases considered above $c = 2|Q_B \setminus \{\mathsf{init}\}|$: if x is locally deadlocked then the Monotonicity Lemma ensures that there is a deadlocked run in $(A, B)^{(1,c)}$. □

Tightness 7 (1-Conj, Deadlocks, Unfair)**.** *The cutoff $c = 2|B| - 2$ is tight for parameterized deadlock detection in the 1-conjunctive systems, i.e., for any k there is a system type (A, B) with $|B| = k$ such that there is a deadlock in $(A, B)^{(1,2|B|-2)}$, but not in $(A, B)^{(1,2|B|-3)}$.*

Proof. Figure 5.7 provides templates (A, B) that proves the observation. In the figure the edge with $\forall\neg b_1, \ldots, \forall\neg b_k$ denotes edges with guards $\forall\neg b_1, \ldots, \forall\neg b_k$. To get the global deadlock we need at least two processes in each $b_i \in \{b_1, \ldots, b_k\}$. Note that the system does not have local deadlocks. □



Template A Template B

Figure 5.7: Templates used to prove Tightness 7

5.8.4 Deadlocks with Fairness: New Constructions

The Monotonicity Lemma is proven by keeping process B_{n+1} in the initial state, and copying the runs of deadlocked processes. If the run of $(A, B)^{(1,n)}$ is globally deadlocked, then process B_{n+1} may keep moving in the constructed run, i.e., the run may be only locally deadlocked. In the case of a local deadlock in $(A, B)^{(1,n)}$, we distinguish two cases: there is an infinitely moving B-process, or all B-processes are deadlocked (and thus A moves infinitely often). In the latter case, we use the same construction as in the global deadlock case (the correctness argument uses the fact that systems are 1-conjunctive, runs are initializing, and there is only one process of type A). In the former case, we copy the original run, and let B_{n+1} share a local run with an infinitely moving B-process.

Lemma 29 (Monotonicity: Conj, Deadlocks, Fair)**.** *For 1-conjunctive systems on strong fair initializing or finite runs:*

$$\forall n \geq 1 : (A,B)^{(1,n)} \text{ has a deadlock } \Rightarrow \ (A,B)^{(1,n+1)} \text{ has a deadlock.}$$

Proof. Let x be a globally deadlocked or locally deadlocked strong-fair initializing run of $(A,B)^{(1,n)}$. We will build a globally deadlocked or locally deadlocked strong-fair initializing run of $(A,B)^{(1,n+1)}$.

If x is finite, then y is the copy of x, and the new process stays in init_B until every process becomes deadlocked, and then is scheduled arbitrarily. Note that y constructed this way may be locally deadlocked rather than globally deadlocked as x is.

Now consider the case when x is locally deadlocked strong-fair initializing.

Let $\mathcal{D}$ be the set of deadlocked B-processes in x, and d be the moment when the processes become deadlocked.

Consider the case $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \neq \emptyset$: copy x into y, and let the new process B_{n+1} wait in init_B and interleave the roles with a process B that moves infinitely often in x, as described in the proof of Lemma 26.

Consider the case $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) = \emptyset$: every B process of $(A,B)^{(1,n)}$ is deadlocked and thus $\mathcal{D} = \mathcal{B}$. Define

$$\mathit{DeadGuards} = \big\{ q \mid \exists B_i \in \mathcal{D} \text{ with a transition guarded "}\forall\neg q\text{" in } (s_d(B_i), e_d(B_i)) \big\}.$$

Note that $Q_A \cap \mathit{DeadGuards} = \emptyset$, because A visits infinitely often init_A and we consider 1-conjunctive systems. Hence, copy x into y, and let the new process B_{n+1} wait in init_B until every process $B_1, ..., B_n$ become deadlocked, and then schedule B_{n+1} arbitrarily. □

As for the Bounding Lemma, we use a construction that is similar to that of properties under fairness for disjunctive systems (Sect. 5.7.2): in the setup phase, we populate some "safe" set of states with processes, and then we extend the runs of non-deadlocked processes to satisfy strong fairness, while ensuring that deadlocked processes never get enabled.

Lemma 30 (Bounding: 1-Conj, Deadlocks, Fair)**.** *For 1-conjunctive systems on strong-fair initializing or finite runs:*

- *with* $c = 2|Q_B \setminus \{\mathsf{init}\}|$ *and any* $n > c$*:*

$$(A,B)^{(1,c)} \text{ has a global deadlock } \Leftarrow \ (A,B)^{(1,n)} \text{ has a global deadlock;}$$

- *with* $c = 2|Q_B \setminus \{\mathsf{init}\}| + 1$ *and any* $n > c$ *(when* $|Q_B| > 2$*):*

$$(A,B)^{(1,c)} \text{ has a local deadlock } \Leftarrow \ (A,B)^{(1,n)} \text{ has a local deadlock;}$$

- *with* $c = 2|Q_B \setminus \{\mathsf{init}\}|$ *and any* $n > c$*:*

$$(A,B)^{(1,c)} \text{ has a deadlock } \Leftarrow \ (A,B)^{(1,n)} \text{ has a deadlock.}$$

Proof. **Global deadlocks.** $c = 2|Q_B \setminus \{\mathsf{init}_B\}|$, see Lemma 28, the fairness does not matter on finite runs.

Local deadlocks. Let $c = 2|Q_B \setminus \{\mathsf{init}_B\}|$. Let $x = (s_1, e_1, p_1)\ldots$ be a locally deadlocked strong-fair intitializing run of $(A, B)^{(1,n)}$ with $n > c$. We construct a locally deadlocked strong-fair initializing run y of $(A, B)^{(1,c)}$.

Let $\mathcal{D}$ be the set of deadlocked processes in x. Let d be the moment in x starting from which every process in $\mathcal{D}$ is deadlocked.

Let $\mathsf{dead}(x)$ be the set of states in which processes $\mathcal{D}$ of $(A, B)^{(1,n)}$ are deadlocked.

Let $\mathsf{dead}_2(x) \subseteq \mathsf{dead}(x)$ be the set of deadlocked states such that: for every $q \in \mathsf{dead}_2(x)$, there is a process $P \in \mathcal{D}$ with $s_d(P) = q$ and that for input $e_{\geq d}(P)$ has a transition guarded with "$\forall\neg q$". Thus, a process in q is deadlocked with $e_d(P)$ only if there is another process in q in every moment $\geq d$.

Let $\mathsf{dead}_1(x) = \mathsf{dead}(x) \backslash \mathsf{dead}_2(x)$. I.e., for any $q \in \mathsf{dead}_1(x)$, there is a process P of $(A, B)^{(1,n)}$ which is deadlocked in $s_d(P) = q$ with input $e_d(P)$, and no transitions from q with input $e_d(P)$ are guarded with "$\forall\neg q$".

Define

$$\mathit{DeadGuards} = \big\{q \mid \exists B_i \in \mathcal{D} \text{ with a transition guarded "}\forall\neg q\text{" in } (s_d(B_i), e_d(B_i))\big\}.$$

Figure 5.8 illustrates properties of sets $\mathit{DeadGuards}$, dead_1, dead_2, $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$.

Let us assume $\mathit{DeadGuards} \neq \emptyset$—the other case is straightforward.

The construction has two phases, the setup and the looping phase.

In the **setup phase**, we copy from x into y:

a. $y(A) = x(A)$;

b. for every $q \in \mathsf{dead}_1$: devote one process of $(A, B)^{(1,c)}$ that copies a process of $(A, B)^{(1,n)}$ deadlocked in q;

c. for every $q \in \mathsf{dead}_2 \setminus \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$: devote two processes of $(A, B)^{(1,c)}$ that copy the behaviour of two processes of $(A, B)^{(1,n)}$ that deadlock in q;

d. for every $q \in \mathsf{dead}_2 \cap \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$: in x, there is a process, $B^{inf}_q \in \mathcal{B}\setminus\mathcal{D}$, that visits q infinitely often, and there is a process, $B^{\perp}_q \in \mathsf{dead}_2$, deadlocked in q. Then:

   1. devote one process of $(A, B)^{(1,c)}$ that copies the behaviour of $B^{\perp}_q$, and
   2. devote one process of $(A, B)^{(1,c)}$ that copies the behaviour of B^{inf}_q until it reaches q at a moment after d, and then provide the same input as $B^{\perp}_q$ receives at moment d. This will deadlock the process;

e. for every $q \in \mathit{DeadGuards} \setminus \mathsf{dead}$: note that $q \in \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$ and, thus, there is a process, $B^{inf}_q \in \mathcal{B}\setminus\mathcal{D}$, that visits q infinitely often. Devote one process of $(A, B)^{(1,c)}$ that copies the behaviour of B^{inf}_q until it reaches q at a moment after d;

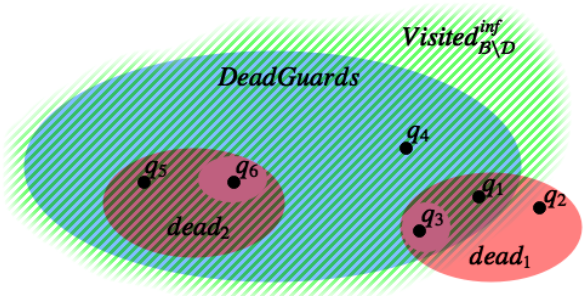


Figure 5.8: Venn diagram for sets $DeadGuards$, dead_1, dead_2, $\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$:

(q_1) $\mathsf{dead}_1 \cap DeadGuards \cap \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \neq \emptyset$ is possible: in x, there is a process deadlocked in state q_1, there is a non-deadlocked process that visits q_1 infinitely often, and there is a process deadlocked in a state $q \neq q_1$ with a transition guarded "$\forall\neg q_1$"

(q_3) $\mathsf{dead}_1 \cap DeadGuards \setminus \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \neq \emptyset$ is possible: similarly to q_1, except that no non-deadlocked processes visit q_3 infinitely often

(q_2) $\mathsf{dead}_1 \setminus (\mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \cup DeadGuards) \neq \emptyset$ is possible: in x, there is a process deadlocked in state q_2, no other processes visit q_2 infinitely often, and no processes are deadlocked with a transition guarded "$\forall\neg q_2$"

(q_4) $DeadGuards \setminus \mathsf{dead} \neq \emptyset$ is possible: there is a process deadlocked in a state $q \neq q_4$ with a transition guarded "$\forall\neg q_4$"

(q_5) $\mathsf{dead}_2 \cap \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \cap DeadGuards \neq \emptyset$ is possible: there is at least one process deadlocked in q_5 with a transition guarded "$\forall\neg q_5$", and some non-deadlocked process visits q_5 infinitely often (this process does not deadlock in q_5, because in q_5 it receives an input different from that of the deadlocked processes)

(q_6) $\mathsf{dead}_2 \cap DeadGuards \setminus \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x) \neq \emptyset$ is possible: similarly to q_5, except no non-deadlocked processes visit q_6 infinitely often

f. if $DeadGuards \setminus \mathsf{dead} \neq \emptyset$ or $A \in \mathcal{D}$, then devote one process that stays in init_B. The process will be used in the looping phase to ensure that the run y is infinite, and that every process of $(A,B)^{(1,c)}$ used in (e) moves infinitely often (and thus y is strong-fair); and

g. let any other process of $(A,B)^{(1,c)}$ (if any) copy behaviour of a process of $(A,B)^{(1,n)}$ that was not used in the construction so far (including this step).

The setup phase ensures: in every state $q \in \mathsf{dead}$, there is at least one process deadlocked in q at moment d in y. Now we need to ensure that the non-deadlocked processes described in steps (e) and (f) move infinitely often, which is done using the looping extension described bellow.

The looping phase is applied to processes in (e) and (f) only[9].

Order arbitrarily $DeadGuards \setminus \mathsf{dead} = (q_1, \ldots, q_k) \subseteq \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$. Note that $\mathsf{init}_B \notin (q_1, ..., q_k)$. Let $\mathcal{P}$ be the set of processes of $(A,B)^{(1,c)}$ used in steps (e) or (f). Note that $|\mathcal{P}| = |(q_1, ..., q_k)| + 1$.

The **looping phase** is: set $i = 1$, and repeat infinitely the following.

- Let $P_{\mathsf{init}} \in \mathcal{P}$ be the process that is currently in init_B, and $P_{q_i} \in \mathcal{P}$ – in q_i.
- Let $B_{q_i} \in \mathsf{Visited}^{inf}_{\mathcal{B}\setminus\mathcal{D}}(x)$ be a process of $(A,B)^{(1,n)}$ that visits q_i and init_B infinitely often. Let P_{init} of $(A,B)^{(1,c)}$ copy transitions of B_{q_i} on some path $\mathsf{init}_B \to \ldots \to g_i$, then let P_{g_i} copy transitions of B_{q_i} on some path $g_i \to \ldots \to \mathsf{init}_B$. For copying we consider only the paths of B_{q_i} that happen after moment d.
- $i = i \oplus 1$.

The number of copies of B that the construction uses in the worst case is (i.e., the item (g) is not used, and we assume $Q_B > 2$, $DeadGuards \setminus \mathsf{dead} = \emptyset$, and $A \in \mathcal{D}$):

$$1_{(f)} + 2|\mathsf{dead}_2|_{(c),(d)} + |\mathsf{dead}_1|_{(b)} \leq 2|Q_B \setminus \{\mathsf{init}_B\}| + 1.$$

Deadlocks. The largest value of c among those for "Local Deadlocks" and for "Global Deadlocks" can be used as the sought value of c for the case of general deadlocks. But it will not be the smallest one. In the proof of the case "Local Deadlocks", in the setup phase, item (e) can be modified for the case when $A \in \mathcal{D}$: since we do not need to ensure that y is infinite, we avoid allocating a process in state init_B. For a given locally deadlocked strong-fair run, the setup phase may produce the globally deadlocked run, but that is allright for the case of general deadlocks. With this note, for the general case $c = 2|Q_B \setminus \{\mathsf{init}_B\}|$. □

Tightness 8 (1-Conj, Deadlocks, Fair)**.** *The cutoff $c = 2|B| - 2$ is tight for deadlock detection on strong-fair initializing or finite runs in the 1-conjunctive systems, i.e., for any $k > 2$ there is a system type (A,B) with $|B| = k$ such that there is a strong-fair initializing deadlocked run in $(A,B)^{(1,2|B|-2)}$, but not in $(A,B)^{(1,2|B|-3)}$.*

Proof. Consider the same templates as in Tightness 7. □

[9]If there are no such processes, then the setup phase produces the sought run y.

5.9 Conclusion

We have extended the cutoffs for guarded protocols of Emerson and Kahlon [37] to support local deadlock detection, fairness assumptions, and open systems. In particular, our results imply the decidability of the parameterized model checking problem for this class of systems and specifications, which to the best of our knowledge was unknown before. Furthermore, the cutoff results can easily be integrated into the parameterized synthesis approach [50].

Since conjunctive guards can model atomic sections and read-write locks, and disjunctive guards can model pairwise rendezvous (for some classes of specifications, see [38]), our results apply to a wide spectrum of systems models. But the expressive power of the model comes at a high cost: cutoffs are linear in the size of a process, and are shown to be tight (with respect to this parameter). For conjunctive systems, our new results are restricted to systems with 1-conjunctive guards, effectively only allowing to model a single shared resource. We conjecture that our proof methods can be extended to systems with more general conjunctive guards, at the price of bigger cutoffs. We leave this extension and the question of finding cutoffs that are independent of the size of processes for future research.

We did preliminary experiments [6] by implementing the synthesizer inside our parameterized synthesizer PARTY [57]. It is a possible future work to find and apply it to real-world applications.

Chapter 6

Parameterized Token Rings

This chapter is based on joint work with R.Bloem and S.Jacobs [54, 57, 20]

Abstract. Parameterized synthesis was recently proposed as a way to circumvent the poor scalability of current synthesis tools. The method uses cutoff results in token rings to reduce the problem to bounded distributed synthesis, and ultimately to a sequence of SMT problems. But experiments show that the size of the specification is a major issue. In this chapter we (1) propose several optimizations of the approach, and (2) perform a parameterized synthesis case study on the industrial arbiter protocol AMBA.

In the first part of this chapter, we optimize the reduction of the parameterized to distributed synthesis. To this end, we refine the cutoff reduction using modularity and abstraction. The evaluation, using our specially developed parameterized synthesizer PARTY, shows that the optimizations lead to several orders of magnitude speed-ups.

In the second part, we perform parameterized synthesis case study on the industrial arbiter protocol AMBA. The AMBA protocol has been used as a benchmark for many reactive synthesis tools, because it is hard to synthesize an implementation that can serve a large number of clients. We show how to use parameterized synthesis to obtain a component that serves a single master, and can be arranged in a ring of arbitrarily many components. We describe new tricks—a cutoff extension tailored for AMBA and decompositional synthesis—that together with the previously described optimizations allowed us to synthesize a component with 14 states in about 1 hour.

6.1 Introduction

By automatically generating correct implementations from a temporal logic specification, reactive synthesis tools can relieve system designers from tedious and error-prone tasks like low-level manual implementation and debugging. This great benefit comes at the cost of high computational complexity of synthesis, which makes synthesis of large systems an ambitious goal. For instance, Bloem et al. [15] synthesize an arbiter for the ARM AMBA Advanced High Performance Bus (AHB) [5].

The results, obtained using RATSY [19], show that both the size of the implementation and the time for synthesis increase steeply with the number of masters that the arbiter can handle. This is unexpected, since an arbiter for $n+1$ masters is very similar to an arbiter for n masters, and manual implementations grow only slightly with the number of masters. While recent results show that synthesis time and implementation size can be improved in standard LTL synthesis tools [46, 48], the fundamental problem of increasing complexity with the number of masters can only be solved by adapting the synthesis approach itself.

To this end, Jacobs and Bloem [50] introduced the *parameterized synthesis* approach. A simple example of a parameterized specification is the following LTL specification of a simple arbiter:

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \wedge \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i). \end{array}$$

In parameterized synthesis, we synthesize a building block that can be cloned to form a system that satisfies such a specification, for any number of components.

Jacobs and Bloem [50] showed that parameterized synthesis is undecidable in general, but semi-decision procedures can be found for classes of systems with cutoffs, i.e., where parameterized verification can be reduced to verification of a system with a bounded number of components. They presented a semi-decision procedure for token-ring networks, building on results by Emerson and Namjoshi [40], which show that for the verification of parameterized token rings, a cutoff of 5 is sufficient for a certain class of specifications. Following these results, parameterized synthesis reduces to distributed synthesis in token rings of (up to) 5 identical processes. To solve the resulting problem, a modification of the SMT encoding of the distributed bounded synthesis problem by Finkbeiner and Schewe [46] was used.

Experiments with the parameterized synthesis method [50] revealed that only very small specifications could be handled with this encoding. For example, the simple arbiter presented before can be synthesized in a few seconds for a ring of size 4, which is the sufficient cutoff for this specification. However, synthesis does not terminate within 2 hours for a specification that also excludes spurious grants, in a ring of the same size. Furthermore, the previously proposed method uses cutoff results of Emerson and Namjoshi [40] and therefore inherits a restricted language support and cannot handle specifications in assume-guarantee style [15]. This precludes the approach from being applied to the AMBA protocol.

In this chapter we address both issues.

In the first part of the chapter (Section 6.4), we optimize the reduction of the parameterized to distributed synthesis. We use the fact that (a) token-ring systems consist of isomorphic processes, (b) different properties may require different cutoffs, and (c) when model checking the behaviours of some fixed processes, the behaviours of the others can be abstracted. The evaluation, using our specially developed parameterized synthesizer PARTY, show that the optimizations lead to several orders of magnitude speed-ups.

In the second part of the chapter (Section 6.5), we perform parameterized synthesis case study on the industrial arbiter protocol AMBA. The AMBA protocol has been used as a benchmark for many reactive synthesis tools, because it is hard

to synthesize an implementation that can serve a large number of clients. We show how to use parameterized synthesis to obtain a component that serves a single master, and can be arranged in a ring of arbitrarily many components. We describe new tricks—a cutoff extension tailored for AMBA and decompositional synthesis—that together with the previously described optimizations allowed us to synthesize a component with 14 states in about 1 hour.

The chapter starts with definitions in Section 6.2, where we introduce token-ring systems, parameterized specifications and problems. Then we state known cutoff results and a slight generalization. Section 6.4 describes the SMT encoding of the bounded synthesis for token-ring systems, followed by optimizations and experiments. Then we proceed to the AMBA case study (Section 6.5). We describe the protocol and its parameterized specification. Section 6.5.2 contains the main contribution: (1) we rewrite the specification into the form feasible to parameterized synthesis and (2) we extend the known cutoffs to handle the resulting AMBA specification. In Section 6.5.3 on experiments, we describe the crucial optimization "decompositional synthesis" and report synthesis timings.

6.2 Definitions

6.2.1 Token-ring Systems

In this section we define token ring systems—the LTS that consists of replicated copies of a process connected in a uni-directional ring. Transitions in a token ring system are either internal or synchronized (in which one process sends the token to the next process along the ring). The token starts in a non-deterministically chosen process.

We start by recalling a (non-deterministic) labeled transition system. A *labeled transition system (LTS)* is a tuple $(I, O, Q, Q_0, \delta, out)$ where I is the set of *inputs*, O is the set of *outputs* disjoint from I, Q is the set of *states*, $Q_0 \subseteq Q$ is the set of *initial states*, $\delta \subseteq Q \times 2^I \times Q$ is the *transition relation*, and $out : Q \to 2^O$ is the *output function* (also called *state-labeling* function).

Fix two disjoint sets: a set $\mathrm{O_{pr}}$ of process template *output variables* that contains two distinguished output variable, snd and tok, and a set $\mathrm{I_{pr}}$ of process template *input variables* that contains a distinguished input variable rcv. We always assume that $\mathrm{I_{pr}}$ and $\mathrm{O_{pr}}$ are disjoint.

Process template. A *process template* P is an LTS $(\mathrm{I_{pr}}, \mathrm{O_{pr}}, Q, Q_0, \delta, out, A_{loc})$:

i) The state set Q is finite and can be partitioned into two non-empty disjoint sets: $Q = T \dot\cup NT$. States in T are said to *have the token*.

ii) The initial state set is $Q_0 = \{\iota_t, \iota_n\}$ for some $\iota_t \in T, \iota_n \in NT$.

iii) The output function is $out : Q \to 2^{\mathrm{O_{pr}}}$ and it satisfies:

- for every $t \in NT$: $\mathsf{tok} \notin out(t)$ and for every $t \in T$: $\mathsf{tok} \in out(t)$,
- for every $t \in Q$: $\mathsf{snd} \in out(t) \to t \in T$.

iv) Let $\Sigma_{\mathsf{pr}} = 2^{\mathrm{I}_{\mathrm{pr}}}$. Let $\Sigma_{\mathsf{pr}}^{\mathsf{rcv}} = \{i \in \Sigma_{\mathsf{pr}} \mid \mathsf{rcv} \in i\}$, $\Sigma_{\mathsf{pr}}^{\neg\mathsf{rcv}} = \Sigma_{\mathsf{pr}} \setminus \Sigma^{\mathsf{rcv}}$, $T^{\mathsf{snd}} = \{q \in Q \mid \mathsf{snd} \in \mathit{out}(q)\}$, $T^{\neg\mathsf{snd}} = T \setminus T^{\mathsf{snd}}$. Then the transition function:

$$\delta \subseteq T^{\mathsf{snd}} \times \Sigma_{\mathsf{pr}}^{\neg\mathsf{rcv}} \times NT \;\cup\; NT \times \Sigma_{\mathsf{pr}}^{\mathsf{rcv}} \times T \;\cup\; NT \times \Sigma_{\mathsf{pr}}^{\neg\mathsf{rcv}} \times NT \;\cup\; T^{\neg\mathsf{snd}} \times \Sigma_{\mathsf{pr}}^{\neg\mathsf{rcv}} \times T. \tag{6.1}$$

Also, δ is non-terminating: for every $q \in NT$ and every $i \in \Sigma_{\mathsf{pr}}$ there exists $q \xrightarrow{i} q'$; and for every $q \in T$ and every $i \in \Sigma_{\mathsf{pr}}^{\neg\mathsf{rcv}}$ there exists $q \xrightarrow{i} q'$.

†) A_{loc} is a *fairness* condition over $\mathrm{I}_{\mathrm{pr}} \cup \mathrm{O}_{\mathrm{pr}}$. We require that on every infinite path from an initial state and satisfying A_{loc}, from any state with the token, $q \in T$, the process reaches a state q' where it sends the token. (In LTL this can be written as $A_{loc} \to \mathsf{G}(\mathsf{tok} \to \mathsf{F}\,\mathsf{snd})$.) We call this requirement (†). We omit A_{loc} in the LTS tuple when it is not important.

Ring topology R. A *ring* is a directed graph $R = (V, E)$, where the set of vertices is $V = \{1, \ldots, k\}$ for some $k \in \mathbb{N}$, and the set of edges is $E = \{(i, i_{mod|V|} + 1) \mid i \in V\}$. We will skip "$mod|V|$" and write $i + 1$. Vertices are called *process indices*.

Token-ring system P^R. Fix a ring topology $R = (V, E)$.

Let $\mathrm{I}_{\mathrm{sys}} = (\mathrm{I}_{loc} \times V) \,\dot\cup\, \mathrm{I}_{glob}$ be the *system input variables*, where local inputs I_{loc} and global inputs I_{glob} are such that $\mathrm{I}_{\mathrm{pr}} = \mathrm{I}_{loc} \,\dot\cup\, \mathrm{I}_{glob}$. For system input $\mathsf{in} \in 2^{\mathrm{I}_{\mathrm{sys}}}$, let $\mathsf{in}(v) = \{i \in \mathsf{in} \mid i \in \mathrm{I}_{loc} \times \{v\} \cup \mathrm{I}_{glob}\}$ denote the input to process v (including global inputs).

Let $\mathrm{O}_{\mathrm{sys}} = \mathrm{O}_{\mathrm{pr}} \times V$ be the *system output variables*. For (p, i) in $\mathrm{O}_{\mathrm{sys}}$ or in $\mathrm{I}_{\mathrm{sys}} \setminus \mathrm{I}_{glob}$ we write p_i.

Given a process template $P = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q, Q_0, \delta, \mathit{out})$ and a token ring topology $R = (V, E)$, the *token-ring system* P^R is the LTS $(\mathrm{I}_{\mathrm{sys}}, \mathrm{O}_{\mathrm{sys}}, S, S_0, \Delta, \mathit{Out})$:

- The set S of *global states* is Q^V, i.e., all functions from V to Q. If $s \in Q^V$ is a global state then $s(i)$ denotes the local state of the process with index i.
- The set of *global initial states* S_0 contains all $s_0 \in Q_0^V$ in which exactly one of the processes has the token.
- The labeling $\mathit{Out}(s) : S \to 2^{\mathrm{O}_{\mathrm{sys}}}$ is: for every $s \in S$: $p_i \in \mathit{Out}(s)$ iff $p \in \mathit{out}(s(i))$, for $p \in \mathrm{O}_{\mathrm{pr}}$ and $i \in V$.

Finally, we define the *global transition relation* Δ. In a *fully asynchronous token ring*, a subset of the processes can make a transition in each step of the system. Thus, Δ consists of the following set of transitions:

- An *internal transition* is an element (s, in, s') of $S \times 2^{\mathrm{I}_{\mathrm{sys}}} \times S$, for which there are process indices $M \subseteq V$ such that
  - i) for all $v \in M$: $\mathsf{snd} \notin \mathit{out}(s(v))$ and $\mathsf{rcv} \notin \mathsf{in}(v)$,
  - ii) for all $v \in M$: $s(v) \xrightarrow{in(v)} s'(v)$ is a transition of P, and
  - iii) for all $u \in V \setminus M$: $s(u) = s'(u)$.

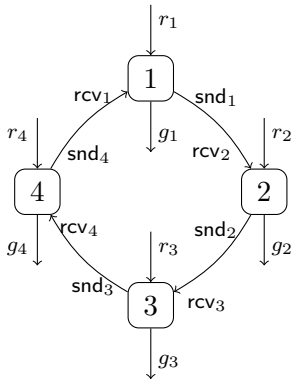


Figure 6.1: Token ring system with 4 processes. Every process has input r and output g. Additionally, every process has input rcv and output snd that are used for passing the token. Thus, $\mathrm{I_{pr}} = \{r, \mathsf{rcv}\}$ and $\mathrm{O_{pr}} = \{g, \mathsf{snd}\}$. In this example, I_{glob} is empty.

- A *token-passing transition* is an element (s, in, s') of $S \times 2^{\mathrm{I_{sys}}} \times S$ for which there are two process indices v and $w = v + 1$ and process indices $M \subset V$ with $\{v, w\} \subseteq M$ such that

  i) $\mathsf{snd} \in out(s(v))$, and $\forall u \in M \setminus \{v\} : \mathsf{snd} \notin out(s(u))$—i.e., only process v sends the token,

  ii) $\mathsf{rcv} \in \mathsf{in}(w)$ and for all $u \in M \setminus \{w\}$: $\mathsf{rcv} \notin \mathsf{in}(u)$—i.e., only process w receives the token,

  iii) for every $u \in M$: $s(u) \overset{in(u)}{\to} s'(u)$ is a transition of P, and

  iv) for every $u \in V \setminus M$: $s'(u) = s(u)$.

Special cases of the fully asynchronous token ring are the *synchronous token ring* and the *interleaving token ring*. In a synchronous token ring, $M = V$ for internal and token-passing transitions, i.e., at each step all the processes simultaneously make a transition. In an interleaving token ring, $M = \{v\}$ for some $v \in V$ for internal transitions, and $M = \{v, w\}$ for $(v, w) \in E$ for token-passing transitions, i.e., at each moment either *exactly one* process makes an internal transition, or one process sends a token to the next process.

An example of processes arranged in a token ring is in Figure 6.1.

System runs. Fix a ring topology $R = (V, E)$ and a process template P. A *run of a token ring system* $P^R = (\mathrm{I_{sys}}, \mathrm{O_{sys}}, S, S_0, \Delta, Out)$ is a maximal-finite or infinite sequence $x = (s_1, \mathsf{in}_1, M_1)(s_2, \mathsf{in}_2, M_2)\ldots$, where:

- $s_1 \in S_0$, $s_k \in S$ and $\mathsf{in}_k \in 2^{\mathrm{I_{sys}}}$ for any $k \leq |x|$,
- for all $k < |x|$: $(s_k, \mathsf{in}_k, s_{k+1}) \in \Delta$,
- for all $k < |x|$: M_k is the set of processes transiting in $(s_k, \mathsf{in}_k, s_{k+1})$ (see M in the definition of Δ).

6.2.2 Parameterized Systems

The *parameterized ring* is the function $\mathcal{R} : n \mapsto \mathcal{R}(n)$, where $n \in \mathbb{N}$ and $\mathcal{R}(n)$ is the ring with n vertices. A *parameterized token-ring system* is a function $P^{\mathcal{R}} : n \mapsto P^{\mathcal{R}(n)}$, where $n \in \mathbb{N}$ and P is a given process template. To disambiguate, we explicitly write "parameterized [fully asynchronous][interleaving][synchronous] token-ring system".

6.2.3 Parameterized Specifications

Parameterized specification is a tuple $\langle \mathrm{I}_{\mathrm{pr}}, \mathrm{I}_{glob}, \mathrm{O}_{\mathrm{pr}}, \Phi \rangle$, where I_{pr} is a set of process template inputs (global and local), I_{glob} is a set of global inputs, O_{pr} is a set of process template outputs, and Φ is an indexed LTL formula over I_{pr} and O_{pr}. Intuitively, an indexed LTL formula is an LTL formula with indexed variables and quantification over indices. Below we define indexed LTL and its sublogic, prenex-indexed LTL.

Indexed LTL

Syntax. Let Vars denote the set of variable names (that will be used as process indices). Let *cond* be a Boolean formula over atoms of the form $x = y$ or $x = y+1$, for arbitrary x, y from Vars. Then an indexed LTL formula Φ over I_{pr}, I_{glob}, and O_{pr} has the grammar:

$$\begin{aligned} \Phi \;=\; & \forall v.(\mathit{cond} \rightarrow \Phi) \mid \exists v.(\mathit{cond} \wedge \Phi) \mid \\ & \Phi \wedge \Phi \mid \neg\Phi \mid \\ & e \mid i_v \mid o_v \mid \Phi \,\mathsf{U}\, \Phi \mid \mathsf{X}_v\, \Phi \end{aligned}$$

where $v \in \mathsf{Vars}$, $i \in \mathrm{I}_{loc}$, $o \in \mathrm{O}_{\mathrm{pr}}$, $e \in \mathrm{I}_{glob}$. We will write $\forall x \neq y : \Phi$ instead of $\forall x \forall y : (x \neq y) \rightarrow \Phi$, and $\exists x \neq y : \Phi$ instead of $\exists x \exists y : x \neq y \wedge \Phi$.

Semantics. We define the semantics for sentence formulas only: a formula Φ is a *sentence* iff every variable v mentioned in the formula is in the scope of a quantifier over that variable. E.g., r_x is not a sentence, while $\forall x : r_x$ is.

Let Φ be a sentence. Let P^R be a token-ring system with $R = (V, E)$ and π be an infinite run of the system. Define $\pi \models \Phi$ iff $\pi \models \Phi_V$ (this satisfaction is defined later), where Φ_V is constructed from Φ as follows.

1. Replace every single-quantified subformula $\forall v.\phi$ of Φ with $\bigwedge_{i \in V} \phi[v \mapsto i]$; replace every single-quantified subformula $\exists v.\phi$ with $\bigvee_{i \in V} \phi[v \mapsto i]$. Here $\phi[v \mapsto i]$ denotes the formula ϕ in which v is substituted by i. E.g., $r_x[x \mapsto 5]$ is r_5.

2. Repeat step (1) until all quantifiers disappear. The resulting formula is Φ_V. Note that conditions *cond* like $x \neq y$ get simplified into true or false.

E.g., $\exists x \exists y . x \neq y \wedge g_x \wedge g_y$ becomes $\bigvee_{(x,y) \in V \times V} . x \neq y \wedge g_x \wedge g_y$.

Definition of "system satisfies Φ". Fix a $P = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q, Q_0, \delta, \mathit{out})$, global inputs I_{glob}, and a token ring $R = (V, E)$. Let Φ be an indexed LTL over I_{pr},

$\mathrm{O_{pr}}$, and I_{glob}. Then $P^R \models \Phi$ iff for every infinite system run π: $\pi \models \Phi$. An infinite system run $\pi = (s_1, in_1, M_1)(s_2, in_2, M_2)... \in (S \times 2^{\mathrm{I_{sys}}} \times 2^V)^\omega$ satisfies Φ iff $(Out(s_1), in_1)(Out(s_2), in_2)... \models \Phi_V$. The latter satisfaction is standard except for the operator X. Given a $v \in V$ and the original run, $(Out(s_1), in_1)(Out(s_2), in_2)... \models \mathsf{X}_v\,\varphi$ iff $(Out(s_i), in_i)(Out(s_{i+1}), in_{i+1})... \models \varphi$ where i is the second[1] smallest i such that $v \in M_i$. Intuitively, $\mathsf{X}_v\,\varphi$ requires φ to hold on the suffix run that skips one transition of the process v and that starts with v transiting. In formulas of the form $\forall i.(...\,\mathsf{X}_i\,...)$, we usually skip the subscript in X_i and write X. (The next operator X_i presented here is inspired by the action-based semantics from [40].)

Prenex-indexed LTL

Let us abbreviate by $\forall x_{cond}.\phi$ the formula $\forall x.cond \rightarrow \phi$, and by $\exists x_{cond}.\phi$ the formula $\exists x.cond \wedge \phi$. When the quantifier is not important, we write $Qx_{cond}.\phi$.

An indexed LTL formula Φ is *prenex-indexed* iff it is of the form

$$Qv^1{}_{cond_{v1}}...Qv^k{}_{cond_{v^k}} : \phi.$$

We call Φ *k-indexed*, because it has k quantifiers. Let LTL\X refer to LTL formulas that do not use X.

Note that prenex-indexed LTL is not as expressive as (non-prenex) indexed LTL. For example, formula $\mathsf{F}\,\forall x.p_x$ does not have an equivalent prenex-indexed form.

Most of existing and our cutoff results are restricted to prenex-indexed LTL formulas with the empty set of global inputs.

Remark 11 ($\forall i.A_i \rightarrow \forall j.G_j$ is not prenex-indexed)**.** In the previous section we defined "a system satisfies an indexed LTL formula". If we use the path quantifier A explicitly, then, as usually, a system satisfies an LTL formula φ, $sys \models \varphi$, is equivalent to $sys \models \mathsf{A}\varphi$, where φ is treated as a path formula of CTL*. Now consider $\forall i.A_i \rightarrow \forall j.G_j$. If rewritten with the path quantifier A, it is $\mathsf{A}(\forall i.A_i \rightarrow \forall j.G_j)$. There is no way to turn it into the form $Qv^1...Qv^k\mathsf{A}\phi$ and this formula is not prenex-indexed.

6.2.4 Parameterized Synthesis Problem

The *parameterized synthesis problem (for token rings)* is:
Given: parameterized specification $\langle \mathrm{I_{pr}}, \mathrm{I}_{glob}, \mathrm{O_{pr}}, \Phi \rangle$
Return: process template $P = (\mathrm{I_{pr}}, \mathrm{O_{pr}}, Q, q_0, \delta, out)$ such that for every n: $P^{\mathcal{R}(n)} \models \Phi$, or "unrealizable" if no such template exists.

We can similarly define the parameterized *model checking* problem, in which the process template is given as input.

Furthermore, we will use the variants of these problems, which ask whether all systems *larger than a given* n_0 satisfy the formula. We call such problems *parameterized*$_{>n_0}$.

[1] Why "second", not the first one? This is the consequence of the fact that we group the input *to be read* with the current output. E.g., $\mathsf{X}_v\,r_v$ should refer to r_v read when transiting *from* the next state rather than referring to r_v read when transiting *into* the next state.

The parameterized synthesis for token rings is undecidable [50], even for prenex 2-indexed specifications without global inputs:

Theorem 31 ([50], Theorem 3.5)**.** *The parameterized synthesis problem of interleaving token rings, without global inputs, formulas $\forall i \neq j.\varphi(i,j)$, is undecidable, where $\varphi(i,j)$ is an LTL$\setminus$X formula over processes i,j.*

This result follows from the undecidability of synthesis of distributed systems with two processes [73]. The problem *is* decidable for prenex 1-indexed specifications.

6.3 Reduction by Cutoffs

The definition of a cutoff is the same as in Section 5.4 on page 72, we repeat it here for completeness. A *cutoff* for parameterized specification $\langle \mathrm{I}_{\mathrm{pr}}, \mathrm{I}_{glob}, \mathrm{O}_{\mathrm{pr}}, \Phi \rangle$ and process template $P = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q, Q_0, \delta, out)$ is a number $c \in \mathbb{N}$ such that

$$\forall n \geq c.\big(P^{\mathcal{R}(c)} \models \Phi \;\Leftrightarrow\; P^{\mathcal{R}(n)} \models \Phi\big).$$

Cutoffs reduce the parameterized synthesis and model checking problems to their non-parameterized variants. E.g., if the cutoff is 2 then the answer to the parameterized$_{>2}$ model checking problem "$\forall n > 2 : P^{\mathcal{R}(n)} \models \Phi$" is the same as the answer to the non-parameterized model checking problem "$P^{\mathcal{R}(2)} \models \Phi$".

Known Cutoffs

In a seminal paper [39, 40] Emerson and Namjoshi proved the following cutoff results.

Theorem 32 ([40])**.** *Let $P = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q, Q_0, \delta, out, A_{loc})$ be a process template, $\mathrm{I}_{glob} = \emptyset$ (no global inputs), $\langle \mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, \Phi \rangle$ a parameterized specification. Assume that the scheduler is interleaving. Then c is a cutoff depending on Φ:*

- $c = 2$ *for* $\forall i.\ \phi(i)$,
- $c = 3$ *for* $\forall i.\forall j_{j=i+1}.\ \phi(i, i+1)$,
- $c = 4$ *for* $\forall i.\forall j_{i\neq j}.\ \phi(i,j)$,
- $c = 5$ *for* $\forall i.\forall j_{i\neq j}.\forall k_{k=i+1}.\ \phi(i, i+1, j)$.

The above cutoff results are restricted to token-ring architectures and do not allow for specifications of the more general k-indexed form. Later in [3] we extended the results to more general networks (directed graphs), where the processes can control the directions in which to send and receive the token, and systems can pass more than one token. The paper also studied k-indexed CTL* properties, also with a bounded alternation depth of path quantifiers.

6.4 Bounded Synthesis of Parameterized Token Rings

6.4.1 SMT Encoding

We encourage the reader to revisit Chapter 2 on page 24 to recall how bounded synthesis works in the case of non-distributed systems. We adapt the encoding to the case of (distributed) token ring systems as follows.

Let us start with SMT constraint about a process template.

SMT constraints for a process template. Let us encode the definition of process template from Section 6.2.1 on page 100:

- Introduce a special output $\mathsf{tok} : T \to \mathbb{B}$ such that $\mathsf{tok}(t)$ holds iff $t \in T$ (recall that we divide the states $Q = T \dot{\cup} NT$). Let us encode Eq. 6.1 (on page 100), which specifies: a process template can send the token only if it has the token; sending the token means a process template loses the token; if a process template receives the token and currently does not have it, then it has the token after the transition.

$$\forall_i.\, \mathsf{G} \left[\begin{array}{l} \mathsf{snd}_i \to \mathsf{tok}_i \\ \mathsf{tok}_i \to (\mathsf{snd}_i \leftrightarrow \mathsf{X}\, \neg \mathsf{tok}_i) \\ \neg \mathsf{tok}_i \to (\mathsf{rcv}_i \leftrightarrow \mathsf{X}\, \mathsf{tok}_i) \end{array} \right] \tag{6.2}$$

- What is left is the condition (†) from the process template definition. We introduce the following LTL formula:

$$\forall i.\ (A_{loc})_i \to \mathsf{G}(\mathsf{tok}_i \to \mathsf{F}\, \mathsf{snd}_i), \tag{6.3}$$

  i.e., a process does not lock the token if the fairness condition A_{loc} is satisfied.

SMT constraints for a system. Now let us encode particularities of (distributed) token-ring systems.

- We compose the system transition function out of process transition functions. Note that all processes share the *same* transition function; the input arguments to the function reflect for what process it is used. To account for scheduling, we introduce additional system inputs $\mathsf{sch}_1, ..., \mathsf{sch}_k$ (where k is the number of processes in a ring), and require that a process $i \in \{1, ..., k\}$ can transit only when sch_i is true (and hence a process does not see its inputs when it is not scheduled).

- The scheduler model (asynchronous/synchronous/interleaving) defines the constraints on the scheduling variables $\mathsf{sch}_1, ..., \mathsf{sch}_k$. For synchronous token rings, all scheduling variables are set to true. For interleaving scheduling, exactly one of the scheduling variables is set to true, except for the token-passing transitions where the two processes transit simultaneously. For asynchronous scheduling, any number (including zero) of the scheduling variables can be true. To specify fair scheduling (for the interleaving or asynchronous

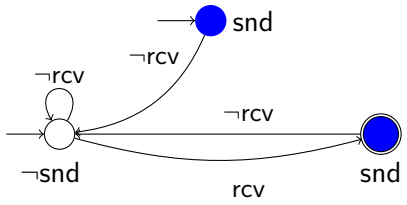


Figure 6.2: Process template synthesized from the specification of a simple arbiter (Example 11). There are two initial states, with and without the token. The blue-filled states have the token, the double state has g, the others have $\neg g$. The process template grants whenever it has the token (except for the initial state), ignoring the request. The exclusivity of the token ensures the mutual exclusion of the grants.

cases), we use the constraint $\bigwedge_i \mathsf{GF}\, sch_i$. This constraint is added as the assumption to the original formula, when we translate the formula into an automaton.

- To ensure that the topology is the token ring (where every process sends the token to its single neighbor), we manipulate process input snd and output rcv in the natural way. For example, if process i is ready to send the token, i.e., it is in a state t and $\mathsf{snd}(t)$ holds, then once it is scheduled we set rcv_{i+1} to true. I.e., snd_i is connected to rcv_{i+1}:

$$\mathsf{G}[\mathsf{snd}_i \leftrightarrow \mathsf{rcv}_{i+1}]$$

Thus, given an LTL formula φ, we want to synthesize a token-ring system that satisfies:

$$\forall i.(\mathsf{GF}\,\mathsf{sch}_i \wedge \mathsf{G}(\mathsf{snd}_i \leftrightarrow \mathsf{rcv}_{i+1})) \;\rightarrow\; \varphi \wedge \forall_i \left(\begin{array}{l} \mathsf{G}[\mathsf{snd}_i \rightarrow \mathsf{tok}_i] \\ \mathsf{G}[\mathsf{tok}_i \rightarrow (\mathsf{snd}_i \leftrightarrow \mathsf{X}\,\neg\mathsf{tok}_i)] \\ \mathsf{G}[\neg\mathsf{tok}_i \rightarrow (\mathsf{rcv}_i \leftrightarrow \mathsf{X}\,\mathsf{tok}_i)] \\ (A_{loc})_i \rightarrow \mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{snd}_i) \end{array} \right) \tag{6.4}$$

Example 11. Consider a specification of a simple arbiter. A process template has inputs $I = \{r, \mathsf{rcv}\}$, outputs $O = \{g, \mathsf{snd}\}$, the original parameterized LTL formula specifying the arbiter is:

$$\begin{array}{rl} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\,g_i) \wedge \neg g_i. \end{array}$$

By Theorem 32, the cutoff is 4. We set $A_{loc} = \mathsf{true}$, instantiate the above formula, and synthesise a token-ring system. The process synthesised using our tool PARTY [57] is in Figure 6.2.

6.4.2 Optimizations

In this section we describe high-level optimizations that are not specific to the SMT encoding. The first two optimizations, *incremental solving* and *modular generation*

of constraints, are sound and complete. The third, *specification strengthening*, is based on automatic rewriting of the specification and introduces incompleteness. The last optimization, *hub-abstraction* is sound and complete.

Incremental Solving

Theorem 32 states that it is sufficient to synthesize a token ring of cutoff size c. However, a solution for a smaller number of processes can still be correct in bigger rings. We propose to proceed incrementally, synthesizing first a ring of size 1, then 2, ..., up to c. After synthesizing a process that works in a ring of size n, we check whether it satisfies the specification also in a ring of size $n+1$. Only if the result is negative, we start to synthesize a ring of size $n+1$.

Modular Constraints for Conjunctive Properties

A useful property of the SMT encoding for parameterized synthesis is that we can separate conjunctive specifications into their parts, generate constraints for the parts separately, and then search for a solution that satisfies the conjunction of all constraints. In the following, for a parameterized specification φ and a number of processes k, let $C(\varphi, k)$ be the set of SMT constraints generated by the bounded synthesis procedure. Note that $C(\varphi, k)$ is of the form $\exists P(...)$. When a process template P is given, let "$P \models C(\varphi, k)$" mean that the constraints $C(\varphi, k)$ are satisfied when instantiated with the process P.

Theorem 33. *Let φ_1 and φ_2 be prenex-indexed formulas such that n_1 is a cutoff for φ_1 and n_2 is a cutoff for φ_2. Then:*

$$P \models C(\varphi_1, n_1) \wedge C(\varphi_2, n_2) \quad \Rightarrow \quad P^{\mathcal{R}(k)} \models \varphi_1 \wedge \varphi_2 \quad \textit{for every } k \geq max(n_1, n_2).$$

The theorem allows us to use different cutoffs for sub-parts of a formula. By conjoining the resulting constraints of all parts, we obtain an SMT problem such that every solution satisfies the complete formula. For example, for a formula

$$\begin{array}{ll} \forall i \neq j. & \mathsf{G}\,\neg(g_i \wedge g_j) \\ \forall i. & \mathsf{G}(r_i \rightarrow \mathsf{F}\, g_i), \end{array}$$

we generate constraints for a ring of size 4 for the first conjunct, and we generated constraints for a ring of size 2 for the second conjunct. This is useful for formulas where the local (1-indexed) part is more complex than the global part, like our more complex arbiter examples.

Specification Strengthening and Handling Assumptions

To handle specifications in assume-guarantee style, we strengthen them in two rewriting steps, which are sound but incomplete. This turns them into the prenex-indexed form (the only form, for which we know how to do parameterized synthesis).

Consider a formula in assume-guarantee style $A_L \wedge A_S \rightarrow G_L \wedge G_S$, where each of the conjuncts is in the prenex-indexed form, and L and S denote respectively

liveness and safety. Notice that this formula, as a whole, is *not* in prenex-indexed form, since it contains process quantifiers inside the path quantifier A (if written explicitly, it says $\mathsf{A}(\forall i... \rightarrow \forall j...)$, which is 2-indexed but not *prenex*-indexed).

Safety-liveness assumptions. Our first strengthening is based on the intuition that often A_L is not needed to obtain G_S, so we strengthen the formula to $(A_S \rightarrow G_S) \wedge (A_L \wedge A_S \rightarrow G_L)$. This step is incomplete for specifications where the system can falsify liveness assumptions A_L and therefore ignore guarantees, or if the assumptions $A_S \wedge A_L$ are unrealizable but A_S is realizable. Both of the cases often hint at the problems with the specification[2].

Localizing assumptions. Consider a 2-indexed formula in assume-guarantee style, $\forall_i A_i \rightarrow \forall_j G_j$, where A_i and G_j refer to process i and j respectively. Originally, we want to plug this formula into Eq. 6.4 and synthesize for the resulting formula. Instead, we localize it—turn $(\forall i...) \rightarrow (\forall j...)$ into $\forall i(... \rightarrow ...)$—and get:

$$\forall i: \ \big(\mathsf{GF}\,\mathsf{sch}_i \wedge \mathsf{G}(\mathsf{snd}_i \leftrightarrow \mathsf{rcv}_{i+1})\big) \rightarrow \left(\begin{array}{l} \mathsf{G}[\mathsf{snd}_i \rightarrow \mathsf{tok}_i] \\ \mathsf{G}[\mathsf{tok}_i \rightarrow (\mathsf{snd}_i \leftrightarrow \mathsf{X}\,\neg\mathsf{tok}_i)] \\ \mathsf{G}[\neg\mathsf{tok}_i \rightarrow (\mathsf{rcv}_i \leftrightarrow \mathsf{X}\,\mathsf{tok}_i)] \\ A_i \rightarrow \mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{snd}_i) \\ A_i \wedge \mathsf{GF}\,\mathsf{tok}_i \rightarrow G_i \end{array}\right) \tag{6.5}$$

A few notes:

- This formula implies the original formula Eq.6.4 where we set $\varphi = \forall_i A_i \rightarrow \forall_i G_i$ and $A_{loc} = A_i$. Setting $A_{loc} = A_i$—requiring $\mathsf{G}(\mathsf{tok}_i \rightarrow \mathsf{F}\,\mathsf{snd}_i)$ to hold under the assumption A_i—is reasonable: it says that if the environment violates the assumption A_i, then we are not required to release the token. Note that if token releasing is required despite A_i, then the rewriting is unsound (it may result in incorrect solutions wrt. Eq.6.4).

- In this formula, the non-technical part (where the technical part encodes the token-ring properties) is in the *prenex*-indexed fragment. Indeed, the non-technical part corresponds to $\forall i.\ (A_i \wedge \mathsf{GF}\,\mathsf{tok}_i \rightarrow G_i)$, which is prenex 1-indexed LTL formula. In contrast, the original formula $\forall i.A_i \rightarrow \forall j.G_j$ is *not* prenex-indexed, because it corresponds to $\mathsf{A}(\forall i.A_i \rightarrow \forall j.G_j)$, if we explicitly write the path quantifier A. Hence for the new formula we can use the cutoff results of Theorem 32, but we could not for the original one.

- Adding $\forall_i\,\mathsf{GF}\,\mathsf{tok}_i$ to the first constraint is crucial. Otherwise, the final formula becomes too restrictive and we may miss solutions. The reason why $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may prevent this is that $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_i$ may work as a local trigger of a

[2]The well known class of GR1 specifications [15], which can be used to describe industrial systems, does not use liveness assumptions for safety guarantees. Furthermore, for GR1 specifications Klein and Pnueli [59] describe a similar separation of safety guarantees from liveness assumptions. They introduce "well separated" assumptions, which are such that the system cannot falsify them at any state, and show that "well separation" of assumptions is sufficient for the rewriting to be sound. Incomplete cases represent specifications where the system can falsify assumptions and ignore guarantees.

violation of an assumption. This is confirmed in the "Pnueli" arbiter experiment, where a violation of one of the assumptions A_i prevents fair token passing in the ring, falsifying $\mathsf{G}\,\mathsf{F}\,\mathsf{tok}_j$ for all $j \neq i$.

- Filiot et al. [45] describe a similar rewriting heuristic, in the context of monolithic synthesis. Our version differs in that we add $\mathsf{GF}\,\mathsf{tok}_i$ assumptions before localization to prevent missing the solutions.

Hub-abstraction

Inspired by the work [30], we introduce the hub abstraction optimization. Recall that for 1-indexed properties $\forall i.\varphi(i)$, a cutoff is 2, meaning that it is enough to consider a token ring system with two processes. Furthermore, by symmetry of the processes, it holds that $P^{\mathcal{R}(2)} \models \forall i.\varphi(i) \Leftrightarrow P^{\mathcal{R}(2)} \models \varphi(1)$ (for details, see [40]). The hub abstraction suggests to replace the process P_2 of a system with the hub process whose whole purpose is to pass the token. This may reduce the state space, because we replace the original process P_2 by the small hub process. We emulate the hub process using the environment assumptions, thus considering only one real process. The assumptions are:

(1) if the process does not have the token, then the environment eventually sends the token (raises the input rcv): $\mathsf{G}(\neg\mathsf{tok} \rightarrow \mathsf{F}\,\mathsf{rcv})$,

(2) if the process has the token, then the environment does not send the token: $\mathsf{G}(\mathsf{tok} \rightarrow \neg\mathsf{rcv})$.

The final formula to synthesize is:

$$\big(\mathsf{GF}\,\mathsf{sch} \wedge \mathsf{G}(\neg\mathsf{tok} \rightarrow \mathsf{F}\,\mathsf{rcv}) \wedge \mathsf{G}(\mathsf{tok} \rightarrow \neg\mathsf{rcv})\big) \;\rightarrow\; \varphi \wedge \left(\begin{array}{l} \mathsf{G}[\mathsf{snd} \rightarrow \mathsf{tok}] \\ \mathsf{G}[\mathsf{tok} \rightarrow (\mathsf{snd} \leftrightarrow \mathsf{X}\,\neg\mathsf{tok})] \\ \mathsf{G}[\neg\mathsf{tok} \rightarrow (\mathsf{rcv} \leftrightarrow \mathsf{X}\,\mathsf{tok})] \\ A_{loc} \rightarrow \mathsf{G}(\mathsf{tok} \rightarrow \mathsf{F}\,\mathsf{snd}) \end{array}\right) \tag{6.6}$$

Note that the assumption (1) states that the token cannot get stuck in the hub process (and thus in the original process that the hub abstracts). This does not always hold, because we only require to pass the token if A_{loc} holds. This means that the hub-abstraction is not sound wrt. Eq.6.4, i.e., there is a process P (and A_{loc}) and $\forall i.\varphi(i)$ such that $P \models$ Eq.6.6 but $P^{\mathcal{R}(2)} \not\models$ Eq.6.4.

However, we will use the hub abstraction in the context of assume-guarantee 1-indexed specifications in the form of Eq.6.5 ("$\forall i. A_i \wedge \mathsf{GF}\,\mathsf{tok}_i \rightarrow G_i$"). Since Eq.6.5 only requires the guarantee to hold on paths where the token is passed infinitely often, the following result holds.

Theorem 34. *For every process template P and LTL formula φ of the form $A \rightarrow G$:*

$$P \models \textit{Eq.6.6 (where } A_{loc} = A) \quad \Rightarrow \quad P^{\mathcal{R}(2)} \models \textit{Eq.6.5.}$$

Furthermore, we can replace $\mathsf{GF}\,\mathsf{sch}$ with *true*, which can introduce unsoundness wrt. Eq.6.5. But this step is sound for formulas where the environment cannot violate guarantees by not scheduling a process. This is true for all examples we consider in the next section.

6.4.3 Evaluating Optimizations

For the evaluation of optimizations we developed an automatic parameterized synthesis tool PARTY [57]. The tool and the benchmarks are available at https://github.com/5nizza/Party/. PARTY

(1) identifies the cutoff of a given LTL specification,

(2) adds token-ring specific guarantees and assumptions to the specification,

(3) translates the modified specification into a UCT using LTL3BA [8],

(4) for a given cutoff and system size bound, builds the SMT constraints,

(5) solves the constraints using SMT solver Z3 v.4.1 [34]. If the solver reports unsatisfiability, then no model for the current bound exists, and the tool goes to step 4 and increases the bound until the user interrupts execution or a model is found. A model synthesized represents a Moore machine that can be copied to form a token-ring system of any size.

We run the experiments on a single core of a Linux machine with two 4-core 2.66 GHz Intel Xeon processors and 64 GB RAM. Reported times in tables include all the steps of the tool. For long running examples, SMT solving contributes most of the time. Timings reported in tables are timings of one particular run, although we observed that the behaviour of optimizations timings does not change much on different runs.

For the evaluation of optimizations we run the tool, with different sets of optimizations enabled, on three examples: a simple arbiter, a full arbiter, and a "Pnueli" arbiter. All benchmarks contain the mutual exclusion property $\forall i \neq j.\, \mathsf{G}\neg g_i \wedge g_j$, for which a cutoff is 4 according to Theorem 32. We show solving times in Table 6.1. The horizontal axis of the table has columns for token rings of different sizes. Each successive optimization below includes previous optimizations.

Incremental solving. Solving times can be sped up considerably by synthesizing a ring of size 2, then checking whether the solution is correct for a ring of size 4. For instance, for the full arbiter, the general solution was found in $\approx$24 seconds when synthesizing a ring of size 2 (time from the "original" row in Table 6.1). Checking if the solution is correct for a ring of size 4 takes additional $\approx$ 30 seconds, thus reducing the synthesis time from more than 2 hours (column "full4" in the same row) to $\approx$ 54 seconds. Times for incremental solving are not given in the table, because its contribution is small when optimizations "strengthening", "modular", and "async hub" are applied.

Strengthening. This version refers to two optimizations described in Section 6.4.2: localizing of assume-guarantee properties and rewriting liveness assumptions from properties with safety guarantees. Formula rewriting significantly reduces the size of the automaton: for example, the automaton corresponding to the "Pnueli" arbiter in a ring of size 4 reduces its size from 1700 to 31 states (from 41 to 16 for the full arbiter).

Table 6.1: Effect of optimizations on synthesis time (in seconds, t/o=2h)

	simple4	full2	full3	full4	pnueli2	pnueli3	pnueli4	pnueli5	pnueli6
original	3	24	934	t/o	23	6737	t/o	t/o	t/o
strengthening	1	6	81	638	2	13	90	620	6375
modular	1	4	8	13	2	4	11	49	262
async hub	1	2	2	5	2	3	9	37	236
sync hub	1	1	2	4	2	3	8	42	191
total speedup	3	20	10^2	$\geq 10^3$	10	10^3	$\geq 10^3$	$\geq 10^2$	≥ 40

Modular. In this version, constraints for formulas of the form $\phi_i \wedge \phi_{i,j}$ are generated separately for local properties ϕ_i and for global properties $\phi_{i,j}$, using the same symbols for transition and output functions. Constraints for ϕ_i are generated for a ring of size 2, and constraints for $\phi_{i,j}$ for a ring of size 4. These sets of constraints are then conjoined in one query and given to the SMT solver. Such separate generation of constraints leads to smaller automata and queries, resulting in approximately 10x speed up.

Hub abstractions. By replacing one of the processes in a ring of size 2 with assumptions on its behavior, we reduce the synthesis of a ring of size two to the synthesis of a single process. In row "async hub" the process is synthesized in an asynchronous setting, while in row "sync hub" the process is assumed to be always scheduled. On these examples, the speed up is insignificant.

6.4.4 Discussion

We showed how optimizations of the SMT encoding, along with modular application of cutoff results, strengthening and abstraction techniques, leads to a significant speed-up of parameterized synthesis. Experimental results show speed-ups of more than three orders of magnitude for some examples.

In the next section, we use these optimizations to tackle AMBA specification. This will not work out of the box and we will introduce more tricks specifically tailored to the AMBA.

6.5 AMBA Protocol Case Study

We demonstrate how to synthesize a parameterized implementation of the AMBA AHB, with guaranteed correctness for any number of masters. To this end, we translate the LTL specification of the AMBA AHB (as found in [51]) into a version that is suitable for parameterized synthesis in token rings, and address several challenges with respect to theoretical applicability and practical feasibility:

- We show how to localize global input and output signals (those that cannot be assigned to one particular master). This is necessary since our approach is based on the replication of components that act only on local information.

- We extend the cutoff results to fully asynchronous timing model and systems with two process templates.
- We describe further optimizations that make synthesis feasible, in particular based on the insight that the AMBA protocol features three different types of accesses, and the control structures for these accesses can be synthesized step-by-step.

6.5.1 Description of the AMBA Protocol

ARMs *Advanced Microcontroller Bus Architecture* (AMBA) [5] is a communication bus for a number of masters and clients on a microchip. One of the crucial parts of AMBA is the *Advanced High-performance Bus* (AHB), a system bus for the efficient connection of processors, memory, and devices.

For convenience, the input signals are depicted in red color and the outputs are blue.

The bus arbiter ensures that only one master accesses the bus at any time. Masters send HBUSREQ to the arbiter if they want access, and receive HGRANT if they are allowed to access it. Masters can also ask for different kinds of *locked transfers* that cannot be interrupted.

The exact arbitration protocol for AMBA is not specified. Our goal is to synthesize a protocol that guarantees safety and liveness properties. According to the specification, any device that is connected to the bus will react to an input with a delay of one time step. I.e., we are considering Moore machines. In the following, we introduce briefly which signals are used to realize the arbiter of this bus for masters.

Requests and grants. The identifier of the master which is currently active is stored in the $n+1$-bit signal HMASTER[n:0], with n chosen such that the number of masters fit into $n+1$ bits. To request the bus, master i raises signal HBUSREQ[i]. The arbiter decides who will be granted the bus next by raising signal HGRANT[i]. When the client raises HREADY, the bus access starts at the next tick, and there is an update HMASTER[n:0] := i, where HGRANT[i] is currently active.

Locks and bursts. A master can request a locked access by raising both HBUSREQ[i] and HLOCK[i]. In this case, the master additionally sets HBURST[1:0] to either SINGLE (single cycle access), BURST4 (four cycle burst) or INCR (unspecified length burst). For a BURST4 access, the bus is locked until the client has accepted 4 inputs from the master (each signaled by raising HREADY). In case of a INCR access, the bus is locked until HBUSREQ[i] is lowered. The arbiter raises HMASTLOCK if the bus is currently locked.

LTL specification. The original natural-language specification [5] has been translated into a formal specification in the GR(1) fragment of LTL before in [51, 15, 48]. Figure 6.3 shows the environment assumptions and system guarantees from [51] that serve as the basis for our parameterized specification. The full specification is $(A1 \wedge \ldots \wedge A4) \rightarrow (G1 \wedge \ldots \wedge G11)$.

Challenges. The AMBA specification has global inputs and outputs (those are without "[i]"), distinguishes 0 from non-0 processes (G10.1 and G11), has the

Assumptions :

$$\mathsf{G}\ (\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR}) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg\text{HBUSREQ}[\text{HMASTER}] \quad (A1)$$

$$\mathsf{G}\,\mathsf{F}\ \text{HREADY} \quad (A2)$$

$$\forall \text{i}:\ \mathsf{G}\ \text{HLOCK}[\text{i}] \rightarrow \text{HBUSREQ}[\text{i}] \quad (A3)$$

$$\forall \text{i}:\ \neg\text{HBUSREQ}[\text{i}] \wedge \neg\text{HLOCK}[\text{i}] \wedge \neg\text{HREADY} \quad (A4)$$

Guarantees :

$$\mathsf{G}\ \neg\text{HREADY} \rightarrow \mathsf{X}\,\neg\text{START} \quad (G1)$$

$$\mathsf{G}\ (\text{HMASTLOCK} \wedge \text{HBURST} = \text{INCR} \wedge \text{START}) \rightarrow \mathsf{X}(\neg\text{START}\ \mathsf{W}\ (\neg\text{START} \wedge \text{HBUSREQ}[\text{HMASTER}])) \quad (G2)$$

$$\mathsf{G}\ (\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \text{HREADY}) \rightarrow \mathsf{X}(\neg\text{START}\ \mathsf{W}\ [3](\neg\text{START} \wedge \text{HREADY})) \quad (G3.1)$$

$$\mathsf{G}\ (\text{HMASTLOCK} \wedge \text{HBURST} = \text{BURST4} \wedge \text{START} \wedge \neg\text{HREADY}) \rightarrow \mathsf{X}(\neg\text{START}\ \mathsf{W}\ [4](\neg\text{START} \wedge \text{HREADY})) \quad (G3.2)$$

$$\forall \text{i}:\ \mathsf{G}\ \text{HREADY} \rightarrow (\text{HGRANT}[\text{i}] \leftrightarrow \mathsf{X}(\text{HMASTER} = \text{i})) \quad (G4)$$

$$\mathsf{G}\ \text{HREADY} \rightarrow (\text{LOCKED} \leftrightarrow \mathsf{X}(\text{HMASTLOCK})) \quad (G5)$$

$$\forall \text{i}:\ \mathsf{G}\ \mathsf{X}\,\neg\text{START} \rightarrow \begin{pmatrix} (\text{HMASTER} = \text{i} \leftrightarrow \mathsf{X}(\text{HMASTER} = \text{i})) \\ \wedge\ (\text{HMASTLOCK} \leftrightarrow \mathsf{X}\,\text{HMASTLOCK}) \end{pmatrix} \quad (G6)$$

$$\forall \text{i}:\ \mathsf{G}\ (\text{DECIDE} \wedge \mathsf{X}\,\text{HGRANT}[\text{i}]) \rightarrow (\text{HLOCK}[\text{i}] \leftrightarrow \mathsf{X}(\text{LOCKED})) \quad (G7)$$

$$\forall \text{i}:\ \mathsf{G}\ \neg\text{DECIDE} \rightarrow \begin{pmatrix} \text{HGRANT}[\text{i}] \leftrightarrow \mathsf{X}\,\text{HGRANT}[\text{i}] \\ \wedge\ \text{LOCKED} \leftrightarrow \mathsf{X}\,\text{LOCKED} \end{pmatrix} \quad (G8)$$

$$\forall \text{i}:\ \mathsf{G}\ \text{HBUSREQ}[\text{i}] \rightarrow \mathsf{F}(\neg\text{HBUSREQ}[\text{i}] \vee \text{HMASTER} = \text{i}) \quad (G9)$$

$$\forall \text{i} \neq 0:\ \mathsf{G}\ \neg\text{HGRANT}[\text{i}] \rightarrow (\neg\text{HGRANT}[\text{i}]\ \mathsf{W}\ \text{HBUSREQ}[\text{i}]) \quad (G10.1)$$

$$\mathsf{G}\ (\text{DECIDE} \wedge (\forall \text{i}: \neg\text{HBUSREQ}[\text{i}])) \rightarrow \mathsf{X}\,\text{HGRANT}[0] \quad (G10.2)$$

$$\text{HGRANT}[0] \wedge (\forall \text{i} \neq 0: \neg\text{HGRANT}[\text{i}]) \wedge \text{HMASTER} = 0 \wedge \neg\text{HMASTLOCK} \wedge \text{DECIDE} \wedge \text{START} \quad (G11)$$

Figure 6.3: Specification of the AMBA AHB [51], in the GR(1) fragment of LTL. The inputs are: HBURST, HBUSREQ[i], HREADY, and HLOCK[i]. The outputs are: HMASTLOCK, HMASTER, START, DECIDE, LOCKED, and HGRANT[i].

assume-guarantee form $\mathsf{A}(\forall i.A1 \wedge ... \wedge A4 \rightarrow \forall i.G1 \wedge ... \wedge G11)$ (thus not in the prenex-indexed form), has the process quantification inside a temporal operator in G10.2 $\mathsf{G}(\forall i... \rightarrow ...)$, and requires a synchronous mode of execution (all processes transit simultaneously). Thus we cannot apply the cutoff results (Theorem 32 on page 104) for parameterized synthesis. The next section shows how to handle this.

6.5.2 Handling the AMBA Specification

This section shows how to rewrite the AMBA specification into a form admissible to the parameterized synthesis. We not only rewrite the specification, but also extend the cutoff results [40] to the resulting class of specifications. Note that the resulting specification is not the same as the original AMBA (but closely resembles it), due to constraints of the token-ring architecture. (For example, token rings cannot ensure immediate granting of a client, because the token has to travel to the corresponding process first.) The resulting specification describes a round-robin arbiter with different granting schemes and one special process.

Special 0-process: two process templates (A, B)

The specification distinguishes between master number 0 and all other masters. We support this by synthesizing two different process implementations, A for the 0-process and B for non-0 processes: the A-process serves master 0 and the B-processes serve the other masters. We denote a token-ring system composed of one A process and n copies of B using the notation $(A, B)^{(1,n)}$. The modified parameterized synthesis problem is to find (A, B) such that $\forall n : (A, B)^{(1,n)} \models \Phi$. Later we will separate the specification into two parts: one will talk about process A, another will talk about B-processes.

Localizing global outputs

The AMBA specification has global outputs HMASTLOCK, HMASTER, START, DECIDE, and LOCKED. They depend on the global state of the system, which is not handled by the work on parameterized model checking of token ring systems [40, 3]. To overcome this, we introduce local versions of the global outputs and build global outputs from them:

- HMASTER $= i$ whenever HMASTER[i] is high, and
- for every global output *glob* from {HMASTLOCK, START, DECIDE, LOCKED}, $glob = \exists i.$ TOK[i] $\wedge\, glob_i$.

We replace each global output with its local version, e.g., START is replaced by START[i]. Note that the limited communication interface (via token passing) does not make AMBA specification unrealizable, although processes cannot access the value of global outputs when they do not possess the token. Intuitively, this is because the token is the shared resource that guarantees mutual exclusion of grants, and therefore the values of these global signals should always be controlled by the process that has the token. In particular, outputs DECIDE and START are used to decide when to raise a grant and when to start and end a bus access[3], which should only be done when the token is present. Similarly, signals HMASTLOCK and HMASTER should be controlled by the process that currently controls the bus (and hence has the token).

Finally, we mentioned many times that the token should be used to ensure the mutual exclusion of grants. Let us explicitly add this requirement into the specification, namely we add G12: $\forall i.$ HGRANT[i] $\rightarrow$ TOK[i]. (The original formula contains only an *implicit* mutual exclusion property: G4 defines how HMASTER is updated by the HGRANT[i] signals, which can only be satisfied if HGRANT[i] are mutually exclusive.)

Splitting the specification into two & other small rewritings

Once we localized global outputs, we can talk about splitting the AMBA specification into two parts. At first, each part will be in the assume-guarantee form,

[3]The original AMBA specification [5] does not have these signals—they were introduced to simplify the formalization of the specification [51].

where the assumptions talk about all the processes (the only A and all B), but the guarantees will be separated into (i) guarantees for the B-processes and (ii) guarantees for the process A.

After the localization, guarantees G10.1 and G10.2 become:

$$\begin{aligned} \forall \mathrm{i} \neq 0 : \mathsf{G} \quad & \neg \mathrm{HGRANT}[\mathrm{i}] \rightarrow (\neg \mathrm{HGRANT}[\mathrm{i}] \,\mathsf{W}\, \mathrm{HBUSREQ}[\mathrm{i}]) \qquad (G10.1) \\ \mathsf{G} \quad & (\mathrm{DECIDE}[0] \wedge \forall \mathrm{i}. \neg \mathrm{HBUSREQ}[\mathrm{i}]) \rightarrow \mathsf{X}\, \mathrm{HGRANT}[0] \quad (G10.2) \end{aligned}$$

Thus, G10.1 is used for B-processes, while G10.2 is used for the process A. Let us talk more about G10.2, because it has two issues.

The first issue with G10.2 is that it requires an immediate reaction to a situation when no process receives a bus request. This is unrealizable in token rings, because mutual exclusion of the grants requires possession of the token, and the token transmission takes time. We modify G10.2 to allow the process A to wait for the token and then immediately react:

$$\mathsf{G}\left(\neg \mathrm{TOK}[0] \wedge \mathsf{X}\, \mathrm{TOK}[0] \wedge \forall \mathrm{i}. \neg \mathrm{HBUSREQ}[\mathrm{i}] \rightarrow \mathsf{X}\, \mathrm{HGRANT}[0]\right).^{4}$$

The second issue with G10.2 is the quantifier $\forall i$ *inside* the temporal operator G (such specifications were not studied in parameterized model checking of token rings). It requires the process A to know about inputs of all B-processes, as it needs to react to a situation where HBUSREQ[i] is low for every process. To get rid of the nesting $\mathsf{G}(\forall i...)$, we introduce a new global input NO_REQ, and add the assumption $\forall i.\, \mathsf{G}(\mathrm{HBUSREQ}[\mathrm{i}] \rightarrow \neg \mathrm{NO\_REQ})$. Then G10.2 becomes:

$$\mathsf{G}\ (\neg \mathrm{TOK}[0] \wedge \mathsf{X}\, \mathrm{TOK}[0] \wedge \mathrm{NO\_REQ}) \rightarrow \mathsf{X}\, \mathrm{HGRANT}[0]).$$

This strengthens the specification, because the environment can set ¬NO_REQ even when there are no requests. This concludes the discussion of G10.2.

The last asymmetric property is the guarantee G11. We split it into two parts:

- G11.1: $\neg \mathrm{HGRANT}[\mathrm{i}] \wedge \neg \mathrm{HMASTLOCK}[\mathrm{i}]$ for B-processes and
- G11.2: $\mathrm{TOK}[0] \rightarrow \mathrm{HGRANT}[0] \wedge \mathrm{HMASTER}[0] \wedge \neg \mathrm{HMASTLOCK}[0]$ (for A).

Localizing global inputs

The AMBA specification in Figure 6.3 uses global inputs HBURST,HREADY, and NO_REQ that we introduced in the previous section.

First, we introduce local versions $\mathrm{HBURST}[i]$, $\mathrm{HREADY}[i]$, and $\mathrm{NO\_REQ}[i]$, and add the assumption $\forall i \neq j.\ loc_i = loc_j$ for $loc \in \{\mathrm{HBURST}, \mathrm{HREADY}, \mathrm{NO\_REQ}\}$.

[4] Maybe you expected to see $\mathsf{G}\ (\mathrm{DECIDE}[0] \wedge (\forall \mathrm{i} : \neg \mathrm{HBUSREQ}[\mathrm{i}]) \wedge \mathsf{X}\, \mathrm{TOK}[0]) \rightarrow \mathsf{X}\, \mathrm{HGRANT}[0])$, but this makes the specification unrealizable due to the token ring requirement $\mathsf{G}(\mathrm{TOK} \rightarrow \mathsf{F}\, \mathrm{SEND})$: the environment falsifies it by making $\forall \mathrm{i} : \neg \mathrm{HBUSREQ}[\mathrm{i}]$ true whenever the process raises DECIDE (and hence the process should continue granting and cannot release the token). To regain realizability one could add additional assumptions (something like $\mathsf{G}\,\mathsf{F}(\mathrm{DECIDE} \wedge \neg \mathrm{NO\_REQ})$). Instead, we decided to change slightly the specification.

This rewriting does not change the specification. The specification becomes

$$\mathsf{A}\Big(\forall i \neq j_{i,j\in\{A,B_1,...,B_n\}}.\Phi(i,j) \;\rightarrow\; \forall k \in \{B_1,...,B_n\}.\Psi(k)\Big) \wedge$$
$$\mathsf{A}\Big(\forall i \neq j_{i,j\in\{A,B_1,...,B_n\}}.\Phi'(i,j) \;\rightarrow\; \Psi'(A)\Big),$$

where each $\Phi(i,j)$ and $\Phi'(i,j)$ talk about propositions of processes i and j, and $\Psi(k)$ and $\Psi'(A)$ talk about propositions of process k and A respectively. Note that $\Psi \neq \Psi'$ because we split the guarantees for B-processes and the process A (the assumptions also slightly differ, so we use Φ and Φ'). Now the specification does neither have global inputs nor global outputs.

Second, we drop the newly introduced assumptions. This means that the original global inputs HBURST, HREADY, and NO_REQ may have different values for different processes, i.e., they are not "global" anymore. This strengthens the specification, because dropping the assumptions enables more environment behaviors (and the formula is universal $\mathsf{A}(...)$). The resulting specification becomes

$$\begin{aligned}&\mathsf{A}\big(\forall i \in \{A,B_1,...,B_n\}.\Phi(i) \;\rightarrow\; \forall j \in \{B_1,...,B_n\}.\Psi(j)\big) \wedge\\ &\mathsf{A}\big(\forall i \in \{A,B_1,...,B_n\}.\Phi'(i) \;\rightarrow\; \Psi'(A)\big).\end{aligned} \tag{6.7}$$

Figures 6.4 and 6.5 define Φ, Ψ, Φ', and Ψ'. We stress that $\Phi(i)$ and $\Phi'(i)$ has to be conjoined over all B-processes *and* the process A to form the assumptions.

Resulting parameterized specification and cutoffs

We still cannot apply the cutoff results to the specification in Eq.6.7, because it is in the assume-guarantee form (and thus is not prenex-indexed) and has the synchronous timing model.

To handle the synchronous timing model, we synthesize a more general case of fully asynchronous systems (those work under all ranges of schedulers from the synchronous to the interleaving one). This represents a more difficult synthesis task, but if the synthesizer finds such a system, then the system works in the synchronous setting too (because we have universal properties).

To handle the assume-guarantee issue, we localize the assumptions as described in Eq.6.5 on page 108, by strengthening $\mathsf{A}(\forall i.\phi(i) \rightarrow \forall j.\psi(j))$ into $\mathsf{A}\big(\forall i.(\phi(i) \wedge \mathsf{GF}\,\mathsf{tok}_i \rightarrow \psi(i))\big)$ where A_i in Eq.6.5 is $\phi(i)$. The final specifications are (in LTL):

$$\boxed{\begin{array}{l}\text{for } B\text{-processes: } \forall i \in \{B_1,...,B_n\}.\Phi(i) \wedge \mathsf{GF}\,\mathsf{tok}_i \rightarrow \Psi(i) \text{ and } A_{loc} = \Phi,\\ \text{for the process } A\text{: } \Phi'(A) \;\rightarrow\; \Psi'(A) \text{ and } A_{loc} = \Phi',\end{array}} \tag{6.8}$$

where Φ, Φ', Ψ, and Ψ' are defined in Figures 6.4 and 6.5 (Recall that A_{loc} is a formula over process propositions such that $\mathsf{G}[\mathsf{tok} \wedge A_{loc} \rightarrow \mathsf{F}\,\mathsf{snd}]$, see definitions on page 99.)

Let us prove cutoffs for specifications of the above form. The parameterized synthesis problem can be separated into two: find (A,B) such that

$$\begin{aligned}\forall n : (A,B)^{(1,n)} \;&\models\; \forall i \in \{B_1,...,B_n\}.\Phi(i) \;\rightarrow\; \Psi(i) \wedge\\ \forall n : (A,B)^{(1,n)} \;&\models\; \Phi'(A) \rightarrow \Psi'(A),\end{aligned}$$

Assumptions $\Phi(i)$:

$$\mathsf{G}\ ((\text{HMASTLOCK}[\mathrm{i}] \wedge (\text{HBURST}[\mathrm{i}] = \text{INCR}) \wedge \text{HMASTER}[\mathrm{i}]) \rightarrow \mathsf{X}\,\mathsf{F}\,\neg\text{HBUSREQ}[\mathrm{i}]) \quad (A1)$$

$$\mathsf{G}\,\mathsf{F}\ \text{READY}[\mathrm{i}] \quad (A2)$$

$$\mathsf{G}\ \text{HLOCK}[\mathrm{i}] \rightarrow \text{HBUSREQ}[\mathrm{i}] \quad (A3)$$

$$\neg\text{HBUSREQ}[\mathrm{i}] \wedge \neg\text{HLOCK}[\mathrm{i}] \wedge \neg\text{READY}[\mathrm{i}] \quad (A4)$$

Guarantees $\Psi(i)$:

$$\mathsf{G}\ \neg\text{READY}[\mathrm{i}] \rightarrow \mathsf{X}\,\neg\text{START}[\mathrm{i}] \quad (G1)$$

$$\mathsf{G}\ (\text{HMASTLOCK}[\mathrm{i}] \wedge \text{HBURST}[\mathrm{i}] = \text{INCR} \wedge \text{START}[\mathrm{i}]) \rightarrow \mathsf{X}(\neg\text{START}[\mathrm{i}]\ \mathsf{W}\ (\neg\text{START}[\mathrm{i}] \wedge \text{HBUSREQ}[\mathrm{i}])) \quad (G2)$$

$$\mathsf{G}\ (\text{HMASTLOCK}[\mathrm{i}] \wedge \text{HBURST}[\mathrm{i}] = \text{BURST4} \wedge \text{START}[\mathrm{i}] \wedge \text{READY}[\mathrm{i}]) \rightarrow \mathsf{X}(\neg\text{START}[\mathrm{i}]\ \mathsf{W}\ [3](\neg\text{START}[\mathrm{i}] \wedge \text{READY}[\mathrm{i}])) \quad (G3.1)$$

$$\mathsf{G}\ (\text{HMASTLOCK}[\mathrm{i}] \wedge \text{HBURST}[\mathrm{i}] = \text{BURST4} \wedge \text{START}[\mathrm{i}] \wedge \neg\text{READY}[\mathrm{i}]) \rightarrow \mathsf{X}(\neg\text{START}[\mathrm{i}]\ \mathsf{W}\ [4](\neg\text{START}[\mathrm{i}] \wedge \text{READY}[\mathrm{i}])) \quad (G3.2)$$

$$\mathsf{G}\ \text{READY}[\mathrm{i}] \rightarrow (\text{HGRANT}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{HMASTER}[\mathrm{i}]) \quad (G4)$$

$$\mathsf{G}\ \text{READY}[\mathrm{i}] \rightarrow (\text{LOCKED}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{HMASTLOCK}[\mathrm{i}]) \quad (G5)$$

$$\mathsf{G}\ \mathsf{X}\,\neg\text{START}[\mathrm{i}] \rightarrow \left(\begin{array}{ll} & \text{HMASTER}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{HMASTER}[\mathrm{i}] \\ \wedge & \text{HMASTLOCK}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{HMASTLOCK}[\mathrm{i}] \end{array}\right) \quad (G6)$$

$$\mathsf{G}\ (\text{DECIDE}[\mathrm{i}] \wedge \mathsf{X}\,\text{HGRANT}[\mathrm{i}]) \rightarrow (\text{HLOCK}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{LOCKED}[\mathrm{i}]) \quad (G7)$$

$$\mathsf{G}\ \neg\text{DECIDE}[\mathrm{i}] \rightarrow \left(\begin{array}{ll} & \text{HGRANT}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{HGRANT}[\mathrm{i}] \\ \wedge & \text{LOCKED}[\mathrm{i}] \leftrightarrow \mathsf{X}\,\text{LOCKED}[\mathrm{i}] \end{array}\right) \quad (G8)$$

$$\mathsf{G}\ \text{HBUSREQ}[\mathrm{i}] \rightarrow \mathsf{F}(\neg\text{HBUSREQ}[\mathrm{i}] \vee \text{HMASTER}[\mathrm{i}]) \quad (G9)$$

$$\mathsf{G}\ \neg\text{HGRANT}[\mathrm{i}] \rightarrow (\neg\text{HGRANT}[\mathrm{i}]\ \mathsf{W}\ \text{HBUSREQ}[\mathrm{i}]) \quad (G10.1)$$

$$\neg\text{HGRANT}[\mathrm{i}] \wedge \neg\text{HMASTLOCK}[\mathrm{i}] \quad (G11.1)$$

$$\mathsf{G}\ \text{HGRANT}[\mathrm{i}] \rightarrow \text{TOK}[\mathrm{i}] \quad (G12)$$

Figure 6.4: Parameterized AMBA specification for B-processes: assumptions $\Phi(i)$ and guarantees $\Psi(i)$. G10.2 is omitted, since it is only needed for the process A.

Assumptions $\Phi'(i)$:

as before: $A1, A2, A3, A4$

new: $\mathsf{G}\ \text{HBUSREQ}[\mathrm{i}] \rightarrow \neg\text{NO\_REQ}[\mathrm{i}]$ $(A6)$

Guarantees $\Psi'(0)$:

as before: $G1, G2, G3, G4, G5, G6, G7, G8, G9, G12$ (where $i = 0$)

removed: $G10.1, G11.1$

modified: $\mathsf{G}(\text{NO\_REQ}[0] \wedge \neg\text{TOK}[0] \wedge \mathsf{X}\,\text{TOK}[0]) \rightarrow \mathsf{X}\,\text{HGRANT}[0])$ $(G10.2)$

modified: $\text{TOK}[0] \rightarrow \text{HGRANT}[0] \wedge \text{HMASTER}[0] \wedge \neg\text{HMASTLOCK}[0]$ $(G11.2)$

Figure 6.5: Parameterized AMBA specification for the process A: modifications wrt. Figure 6.4. The index 0 denotes the process A.

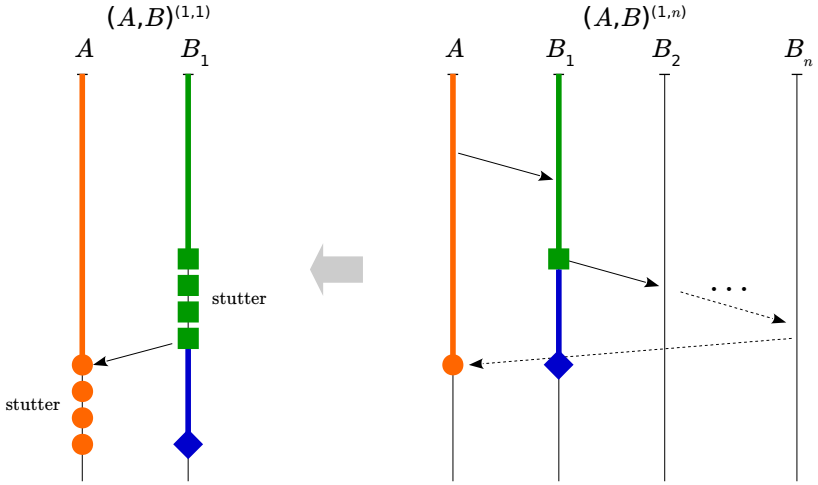


Figure 6.6: Constructing a run of a cutoff system from a run of a large system. Vertical lines depict (local) paths of the processes, the horizontal lines mean the token transmission. The process A starts with the token.

where A_{loc} is either Φ or Φ'.

Theorem 35. *Given two process templates,* $A = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q^A, Q_0^A, \delta^A, out^A, A_{loc}^A)$ *and* $B = (\mathrm{I}_{\mathrm{pr}}, \mathrm{O}_{\mathrm{pr}}, Q^B, Q_0^B, \delta^B, out^B, A_{loc}^B)$*, and let* $\mathrm{I}_{glob} = \emptyset$ *(no global inputs). Assume that initially the process* A *has the token. Then a cutoff is* $(1,1)$ *(one* A*-process and one* B*-process) for the following PMCPs:*

(1) $\forall n : (A,B)^{(1,n)} \models A_{loc}^A \wedge \mathsf{GF}\,\mathsf{tok}_A \;\rightarrow\; \psi(A)$,

(2) $\forall n : (A,B)^{(1,n)} \models \forall i : A_{loc,i}^B \wedge \mathsf{GF}\,\mathsf{tok}_i \;\rightarrow\; \psi(B_i)$.

where $A_{loc,i}^B$ *is* A_{loc}^B *with all propositions subscripted with* i*,* $\psi(p)$ *is an LTL formula over propositions of a process* $p \in \{A, B_1, ..., B_k\}$.

Proof idea. The proof is inspired by the original proof [40].

Item (1). Fix an arbitrary $n > 1$ and let $\varphi(A) = A_{loc}^A \wedge \mathsf{GF}\,\mathsf{tok}_A \rightarrow \psi(A)$. We prove that

$$(A,B)^{(1,1)} \models \varphi(A) \quad \Leftrightarrow \quad (A,B)^{(1,n)} \models \varphi(A).$$

Consider direction $\Rightarrow$. After contra-positioning:

$$(A,B)^{(1,1)} \not\models \varphi(A) \quad \Leftarrow \quad (A,B)^{(1,n)} \not\models \varphi(A).$$

Given a system run of $(A,B)^{(1,1)}$ that satisfies $\neg\varphi(A)$, we build a system run of $(A,B)^{(1,n)}$ that satisfies $\neg\varphi(A)$. The construction is in Figure 6.6. We copy the behaviors of processes A and B_1 until before B_1 sends the token. At this moment, we postpone sending the token by B_1 and stutter[5] it, while the process A continues execution until it gets into state ● ready to receive the token. Then B_1 transmits the token to process A. After that we move process B_1 into state

[5] To "stutter a process p" means "not to schedule it". As a result, a stuttered process neither reads inputs nor changes its state. In the figures it is shown by repeating a state.

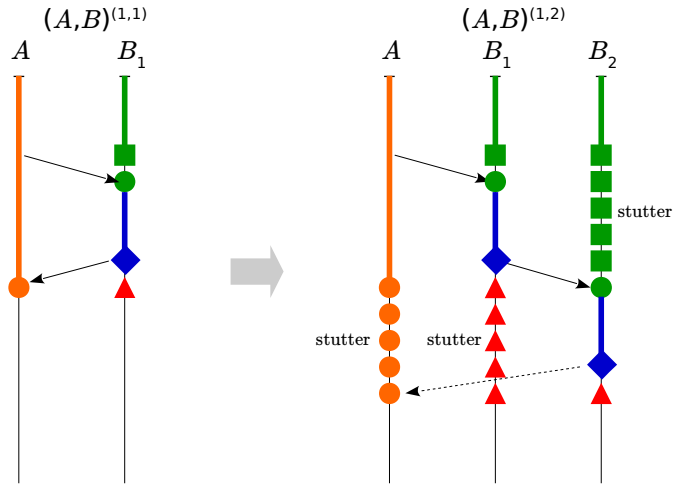


Figure 6.7: Constructing a run of a system $(A, B)^{(1,2)}$ from a run of a cutoff system $(A, B)^{(1,1)}$. Vertical lines depict (local) paths of the processes, the horizontal lines mean the token transmission. The process A starts with the token.

♦, while A stutters in ●. Now we are in the original situation and repeat the construction. Since the property talks about process A only, the resulting run satisfies it. Finally, we assumed that the processes of the large system pass the token infinitely often. If some process $B_x \in \{B_1, ..., B_n\}$ holds the token forever, then we use its behavior for B_1 in the cutoff system (this may require to insert stuttering steps into behaviors of B_1 and A of the cutoff system, to synchronize their (finitely many) token transmissions).

Consider direction $\Leftarrow$. After contra-positioning:

$$(A, B)^{(1,1)} \not\models \varphi(A) \quad \Rightarrow \quad (A, B)^{(1,n)} \not\models \varphi(A).$$

Given a system run of $(A, B)^{(1,n)}$ that satisfies $\neg\varphi(A)$, we build a system run of $(A, B)^{(1,1)}$ that satisfies $\neg\varphi(A)$. Figure 6.7 shows how to construct a run of a system that has one more B process than the cutoff system. By repeating the construction we can add the necessary $n-1$ B-processes. The construction works as follows. The new process B_2 copies the behavior of B_1 until before B_1 receives the token (i.e., up to the state ■). Then it stutters in ■ awaiting for the token from process B_1. After that it copies B_1 behavior from state ● till ♦, while processes A and B_1 stutter. Then B_2 sends the token to A and we return to the original situation. Finally, the case of B_1 or A holding the token forever is straightforward.

Item (2). Consider the case $\forall i.\varphi(B_i)$. First, we use the symmetry argument: for every n,

$$(A, B)^{(1,n)} \models \forall i.\varphi(B_i) \;\Leftrightarrow\; (A, B)^{(1,n)} \models \varphi(B_1).$$

It holds because, for every B_i and system run that satisfies $\neg\varphi(B_i)$, we can construct a run that satisfies $\neg\varphi(B_1)$. The latter is possible because all B-processes start without the token and φ is 1-indexed[6].

[6]In contrast, the symmetry argument will not work for properties of the form $\forall i.\varphi(A, B_i)$, because B_1, B_n, and $B_{x\in\{2,...,n-1\}}$ have different "relation" to A. For example, take the formula

After applying the symmetry argument, we can use the very same constructions as in item (1), see Figures 6.6 and 6.7. Let us only note the case when the token is stuck in some process. As for the construction in Figure 6.7, this is simple: the token will be stuck in A or in B_1 in the large system too. Consider the case in Figure 6.6, when the token gets stuck in some process B_d for $d \neq 1$. This is the only place in the proof where we use the peculiar structure of the formula to verify: $A^B_{loc,1} \wedge \mathsf{GF}\,\mathsf{tok}_1 \rightarrow \psi(B_1)$. Recall that the contra-position negates it and gives $A^B_{loc,1} \wedge \mathsf{GF}\,\mathsf{tok}_1 \wedge \neg\psi(B_1)$. Thus, in the large system the process B_1 receives the token infinitely often, and we can simply ignore the case[7]. □

Let us note that without the assumption "A starts with token" the constructions break. We conjecture that in this case a cutoff increases to $(1, 2)$.

6.5.3 Experiments

In this section, we describe optimizations that are crucial for the synthesis of the parameterized AMBA, and present synthesis timings and resulting implementations. Most of the optimizations were already described in Section 6.4.2. One interesting and not previously described optimization is "Decompositional synthesis", where the specification is synthesized incrementally, starting from a subset of the properties. It is this optimization that allowed us to synthesize the AMBA.

Prototype. Our prototype is based on our tool Party [57], a synthesizer of parameterized token rings. Party is written in Python, uses LTL3BA [8] for automata translation and Z3 [34] for SMT solving. The prototype and specification files can be found at https://github.com/5nizza/Party/ (branch 'amba-gr1'). The experiments were run on a x86_64 machine with 2.6GHz CPU, 12GB RAM, Ubuntu OS.

Synchronous hub abstraction (Section 6.4.2). Synchronous hub abstraction can be applied to 1-indexed specifications. It lets the environment simulate all but one process, and always schedules this process. Thus, the synthesizer searches for a process template in the synchronous setting with additional assumptions on the environment, namely: (i) the environment sends the token to the process infinitely often, and (ii) the environment never sends the token to the process if it already has it. The synchronous hub abstraction is *sound and complete* for 1-indexed properties. After applying this optimization *any* monolithic synthesis method can be applied to the resulting specification (in the form of Eq.6.6 on page 109).

Hardcoding states with and without the token [54, Section4]. The number of states with and without the token in a process template defines the degree of the parallelism in a token ring. Parallelism increases with the number of states that do not have the token. In the AMBA case study, any grants related action depends on having the token. Thus we divide the states in the process template: (a) one state does not have the token, while (b) all other states have the token. We do this by hardcoding the TOK output function.

$\forall i.\, \mathsf{G}(\mathsf{tok}_A \rightarrow \mathsf{tok}_A\,\mathsf{W}\,\mathsf{tok}_i)$. The (wrongly applied) symmetry argument would produce $\mathsf{G}(\mathsf{tok}_A \rightarrow \mathsf{tok}_A\,\mathsf{W}\,\mathsf{tok}_1)$, which says that the token moves from A to B_1 (trivially true in every system), but the original formula does not hold.

[7] We *did* consider the case in other proof branches, to avoid relying on the peculiarity of the formula. We conjecture that in the case of (more general) properties of the form $\forall i.\varphi(B_i)$ (without $\mathsf{GF}\,\mathsf{tok}_i$), the cutoff increases to $(1, 2)$.

Addit. assumptions	time	#states
G HLOCK G HBURST = BURST4	16min.	10
G HBURST = BURST4	13sec.	13
– (Full Specification)	1min.	14

Table 6.2: Results for non-0 process.

Addit. assumptions	time	#states
G HLOCK G HBURST = BURST4	3h.	11
G HBURST = BURST4	1min.	11
– (Full Specification)	1m30s.	12

Table 6.3: Results for 0-process (bursts reduced: $3/4 \rightarrow 2/3$).

Decompositional synthesis of different grant schemes. The idea of the decompositional synthesis is: synthesize a subset of the properties, then synthesize a larger subset using the model from the previous step as the basis. Consider an example of the synthesis of the non-0-process of AMBA. The flow is:

1. Assume that every request is a locked request of type BURST4, i.e., add the assumption G(HLOCK[i] $\wedge$ HBURST = BURST4) to the specification. This implicitly removes guarantee G2 and assumption A1 from the specification. Synthesize the model. The resulting model has 10 states (states $t0, .., t9$ and transitions between them in Figure 6.8).

2. Use the model found in the previous step as the basis: assert the number of states, values of output functions in these states, transitions for inputs that satisfy the previous assumption. Transitions for inputs that violate the assumption from step 1 are not asserted, and thus are left to be synthesized.

   Now relax the assumptions: allow locked and non-locked BURST4 requests, i.e., replace the previous assumption with G(HBURST = BURST4). Again, this implicitly removes G2 and A1. In contrast to the last step, now guarantee G3 is not necessarily 'activated' if there is a request.

   Synthesize the model. This may require increasing the number of states (and it does, in the case of non-0 process)—add new states and keep assertions on all the previous states.

3. Assert the transitions of the model found, as in the previous step.
   Remove all added assumptions and consider the original specification. Synthesize the final model.

Although for AMBA this approach was successful, it is not clear how general it is. For example, it does not work if we start with locked BURST4 and HREADY always high, and then try to relax it. Also, the separation into sets of properties to be synthesized was done manually.

Results. Synthesis times are in Tables 6.2 and 6.3, the model synthesized for non-0 process is in Figure 6.8. The table has timings for the case when all optimizations described in this section are enabled — it was not our goal to evaluate the optimizations separately, but to find a combination that works for the AMBA case study.

For the 0-process we considered a simpler version with burst lengths reduced to 2/3 instead of the original 3/4 ticks. With the original length, the synthesizer could not find a model within 2 hours (it hanged checking 11 state models, while the model has at least 12 states).

Without the decompositional approach, the synthesizer could not find a model for for non-0 process of the AMBA specification within 5 hours.

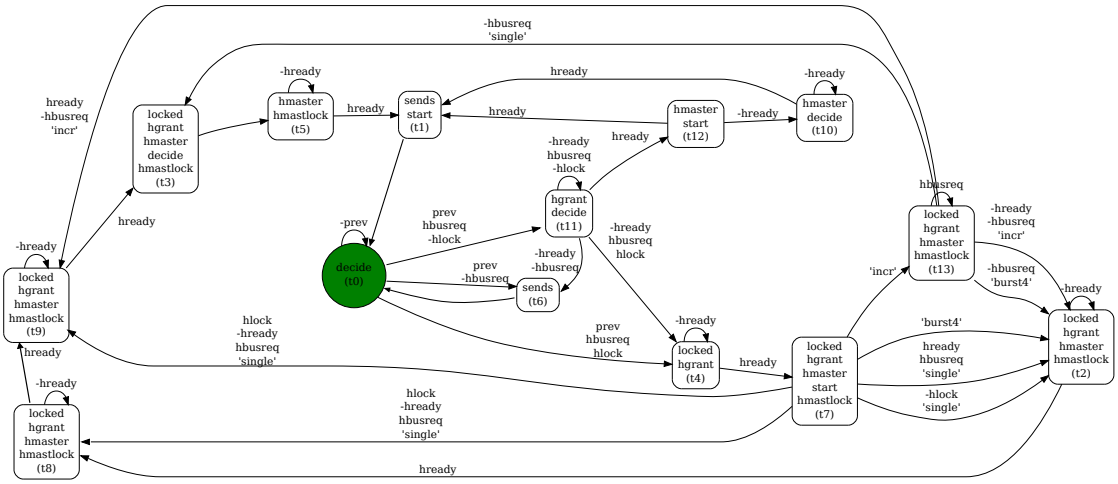


Figure 6.8: Synthesized model of non-0-processes (after manual simplification). Circle green state ($t0$) is without the token, other states are with the token. The initial state is $t0$. States are labeled with their active outputs. Edges are labeled with inputs, a missing input variable means "don't care". 'Burst4' means HBURST = BURST4, 'incr' means HBURST = INCR, 'single' means neither of them. In the first step of decompositional synthesis states $t0, .., t9$ were synthesized, in the second $t10, .., t12$ were added, in the final step state $t13$ was added.

6.5.4 Discussion

We have shown that parameterized synthesis in token rings can be used to solve benchmark problems of significant size, in particular the well-known AMBA AHB specification that has been used as a synthesis benchmark for a long time. To achieve this goal, we slightly extended the cutoff results that parameterized synthesis is based on, and used a number of optimizations in the translation of the specification and the synthesis procedure itself to make the process feasible.

This is the first time that the AMBA case study, or any other realistic case, has been solved by an automatic synthesis procedure for the parameterized case. However, some of the steps in the procedure are manual or use an ad-hoc solution for the specific problem at hand, like the limited extension of cutoff results for global inputs, the construction of suitable functions to convert local to global outputs, or the decompositional synthesis for different grant schemes. Generalizing and automating these approaches is a possible future work.

Our synthesized implementation is such that the size of the parallel composition grows only linearly with the number of components. Thus, for this case study our approach does not only solve the problem of increasing synthesis time for a growing number of components, but also the problem of implementations that need an exponential amount of memory in the number of components. We pay for this small amount of memory with a less-than-optimal reaction time, as processes have to wait for the token in order to grant a request. This restriction could be remedied by extending the parameterized synthesis approach to different system models, e.g., processes that coordinate by guarded transitions [38], or communicate via broadcast messages [44].

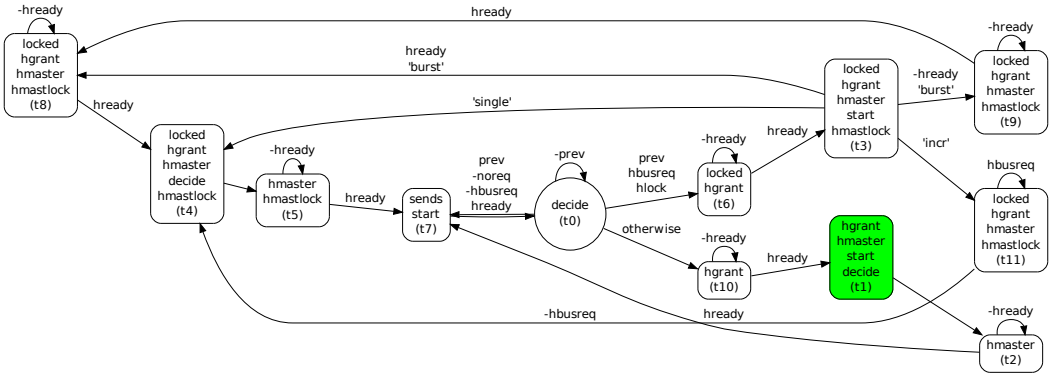


Figure 6.9: Synthesized model of 0-processes (after manual simplification). Circle state ($t0$) is without the token, other states are with the token. The initial state is $t1$. States are labeled with their active outputs. Edges are labeled with inputs, a missing input variable means "don't care". 'Burst' means HBURST = BURST3 (recall we decreased the length of bursts for 0 process), 'incr' means HBURST = INCR, 'single' means neither of them. In the first step of decompositional synthesis states $t0, .., t10$ were synthesized, in the second only transitions were synthesized, but no new states added, in the final step $t11$ was added.

6.6 Conclusion

In this chapter, we studied the parameterized synthesis of token-ring systems from the applied perspective. The starting point was the original approach of Bloem and Jacobs [50], which could be applied only to toy specifications. We suggested several optimizations that made it applicable to larger "made-up" specifications. Then we tackled the real-life specification, that of the AMBA bus protocol, and suggested further optimizations. This required us to extend the theory behind the approach. In the end, we synthesized a solution for the AMBA specification in the parameterized sense, for the first time ever.

Bibliography

[1] SYNTCOMP. http://www.syntcomp.org/, 2017.

[2] Rajeev Alur, Thomas Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *Journal of the ACM*, pages 100–109. IEEE Computer Society Press, 1997.

[3] B. Aminof, S. Jacobs, A. Khalimov, and S. Rubin. Parameterized model checking of token-passing systems. In *VMCAI*, volume 8318 of *LNCS*, pages 262–281. Springer, 2014.

[4] B. Aminof, T. Kotek, S. Rubin, F. Spegni, and H. Veith. Parameterized model checking of rendezvous systems. In *CONCUR*, volume 8704 of *LNCS*, pages 109–124. Springer, 2014.

[5] ARM Ltd. AMBA specification (rev.2). Available from www.arm.com, 1999.

[6] S. Außerlechner. Parameterized Synthesis of Guarded Systems (Master Thesis). *TU Graz Library*, May 2015. Available at https://diglib.tugraz.at/download.php?id=576a77d1edae0&location=browse.

[7] Simon Außerlechner, Swen Jacobs, and Ayrat Khalimov. Tight cutoffs for guarded protocols with fairness. In Barbara Jobstmann and K. Rustan M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 476–494. Springer, 2016.

[8] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS*, volume 7214 of *LNCS*, pages 95–109. Springer, 2012.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[10] Nathalie Bertrand, John Fearnley, and Sven Schewe. Bounded Satisfiability for PCTL. In Patrick Cégielski and Arnaud Durand, editors, *CSL*, volume 16 of *LIPICS*, pages 92–106, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. *SIGPLAN Not.*, 49(1):221–233, January 2014.

[12] Tewodros Awgichew Beyene. *Temporal Program Verification and Synthesis as Horn Constraints Solving*. PhD dissertation, Technical University of Munich, 2015.

[13] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[14] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification.

[15] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[16] Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In Swarat Chaudhuri and Azadeh Farzan, editors, *CAV*, volume 9779 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2016.

[17] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In Christel Baier and Cesare Tinelli, editors, *TACAS*, pages 517–532, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[18] Roderick Bloem, Hana Chockler, Masoud Ebrahimi, and Ofer Strichman. Synthesizing non-vacuous systems. In Ahmed Bouajjani and David Monniaux, editors, *VMCAI*, pages 55–72, Cham, 2017. Springer International Publishing.

[19] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy - a new requirements analysis tool with synthesis. In *CAV*, volume 6174 of *LNCS*, pages 425–429. Springer, 2010.

[20] Roderick Bloem, Swen Jacobs, and Ayrat Khalimov. Parameterized synthesis case study: AMBA AHB. In *SYNT*, volume 157 of *EPTCS*, pages 68–83, 2014.

[21] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, September 2015. 170 pages.

[22] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.

[23] Roderick Bloem, Sven Schewe, and Ayrat Khalimov. CTL* synthesis via LTL synthesis. In *SYNT Workshop*. EPTCS, 2017.

[24] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of parametric concurrent systems with prioritised FIFO resource management. *Formal Methods in System Design*, 32(2):129–172, 2008.

[25] J. R. Büchi and L. H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[26] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 252–263. ACM, 2017.

[27] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[28] Alonzo Church. Logic, arithmetic, and automata. In *International Congress of Mathematicians (Stockholm, 1962)*, pages 23–35. Institute Mittag-Leffler, Djursholm, 1963.

[29] E. M. Clarke, M. Talapur, and H. Veith. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, volume 4963 of *LNCS*, pages 33–47. Springer, 2008.

[30] E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR*, volume 3170 of *LNCS*, pages 276–291. Springer, 2004.

[31] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[32] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[33] Emanuele De Angelis, Alberto Pettorossi, and Maurizio Proietti. Synthesizing concurrent programs using answer set programming. *Fundamenta Informaticae*, 120(3-4):205–229, 2012.

[34] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[35] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[36] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[37] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000.

[38] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE Computer Society, 2003.

[39] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. Principles of Programming Languages*, pages 85–94, 1995.

[40] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *Foundations of Computer Science*, 14:527–549, 2003.

[41] E. Allen Emerson and Joseph Y. Halpern. 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic. *J. ACM*, 33(1):151–178, January 1986.

[42] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, September 1999.

[43] E. Allen Emerson and A. Prasad Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175 – 201, 1984.

[44] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.

[45] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Form. Methods Syst. Des.*, 39(3):261–296, 2011.

[46] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[47] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[48] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. Synthesis of amba ahb from formal specification: a case study. *STTT*, 15(5-6):585–601, 2013.

[49] S. Jacobs and R. Bloem. Parameterized synthesis. In *TACAS*, volume 7214 of *LNCS*, pages 362–376. Springer, 2012.

[50] S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science*, 10:1–29, 2014.

[51] Barbara Jobstmann. *Applications and Optimizations for LTL Synthesis*. PhD thesis, Graz University of Technology, 2007.

[52] Marcin Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[53] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010.

[54] A. Khalimov, S. Jacobs, and R. Bloem. Towards efficient parameterized synthesis. In *VMCAI*, volume 7737 of *LNCS*, pages 108–127. Springer, 2013.

[55] Ayrat Khalimov. Specification format for reactive synthesis problems. In *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015.*, pages 112–119, 2015.

[56] Ayrat Khalimov and Roderick Bloem. Bounded synthesis for streett, rabin, and ctl*. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 333–352. Springer, 2017.

[57] Ayrat Khalimov, Swen Jacobs, and Roderick Bloem. Party parameterized synthesis of token rings. In *Computer Aided Verification*, pages 928–933. Springer, 2013.

[58] P. Klampfl, R. Koenighofer, R. Bloem, A. Khalimov, A. Abu-Yonis, and S. Moran. OpenSEA: Semi-Formal Methods for Soft Error Analysis. *ArXiv e-prints*, December 2017.

[59] Uri Klein and Amir Pnueli. Revisiting synthesis of gr(1) specifications. In Sharon Barner, Ian G. Harris, Daniel Kroening, and Orna Raz, editors, *Haifa Verification Conference*, volume 6504 of *Lecture Notes in Computer Science*, pages 161–181. Springer, 2010.

[60] Tobias Klenze, Sam Bayless, and Alan J Hu. Fast, flexible, and minimal CTL synthesis via SMT. In *International Conference on Computer Aided Verification*, pages 136–156. Springer, 2016.

[61] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *FOCS*, pages 531–542, 2005.

[62] Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. *Bulletin of Symbolic Logic*, 5(2):245–263, 1999.

[63] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, March 2000.

[64] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Inf. and Comp.*, 117(1):1–11, 1995.

[65] Z. Manna and A. Pnueli. Temporal specification and verification of reactive modules. *Weizmann Institute of Science Technical Report*, 1992.

[66] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 253–281. Springer, 1981.

[67] David E Muller and Paul E Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of rabin, mcnaughton and safra. *Theoretical Computer Science*, 141(1-2):69–107, 1995.

[68] Nir Piterman. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. *Logical Methods in Computer Science*, Volume 3, Issue 3, August 2007.

[69] Nir Piterman and Amir Pnueli. Faster solutions of Rabin and Streett games. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 275–284, 2006.

[70] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[71] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190. ACM Press, 1989.

[72] Amir Pnueli and Roni Rosner. *On the synthesis of an asynchronous reactive module*, pages 652–671. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989.

[73] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 746–757. IEEE Computer Society, 1990.

[74] Nicola Prezza. CTL (computation tree logic) SAT solver.

[75] M.O. Rabin. *Automata on Infinite Objects and Church's Problem.* Number 13 in Conference Series in Mathematics. American Mathematical Society, 1969.

[76] Roni Rosner. *Modular synthesis of reactive systems.* PhD thesis, PhD thesis, Weizmann Institute of Science, 1992.

[77] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988.

[78] Sven Schewe. Tighter bounds for the determinisation of Büchi automata. In *Proceedings of the Twelfth International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2009), 22–29 March, York, England, UK*, volume 5504 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 2009.

[79] I. Suzuki. Proving properties of a ring of finite state machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

[80] M Y Vardi and L Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 240–251, New York, NY, USA, 1985. ACM.

[81] Moshe Y Vardi. Branching vs. linear time: Final showdown. In *TACAS*, volume 1, pages 1–22. Springer, 2001.

[82] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths (extended abstract). In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 185–194. IEEE Computer Society, 1983.