Topics in Formal Synthesis and Modeling

by

Uri Klein

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

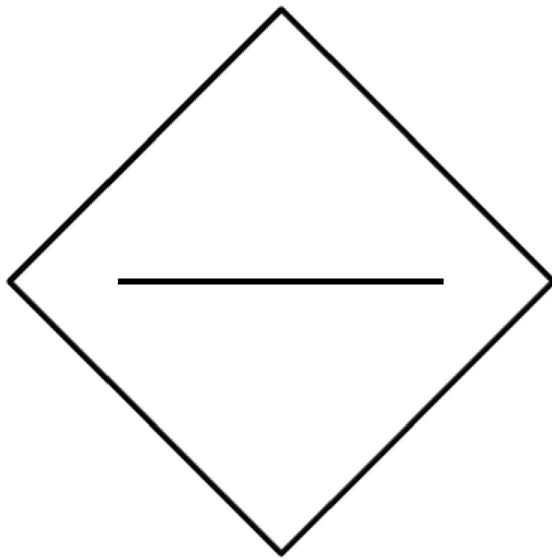
September, 2011

Amir Pnueli

Lenore Zuck

© Uri Klein

All Rights Reserved, 2011



...

In memory of Amir Pnueli

Acknowledgments

First and foremost, I would like to thank my advisor Amir Pnueli who introduced me to the world of formal methods, and who inspired me with his endless optimism and creativity. Undoubtedly, my favorite moments of the PhD were those when we brainstormed and chatted in his office. I had the honor of being Amir's last student, and I am grateful for the opportunity to meet him and to work with him.

Many thanks to my advisor Lenore Zuck who picked me up after Amir's sudden passing away, and who helped me get back on my feet and continue working.

I would also like to thank Nir Piterman, who functioned as an unofficial research advisor to me in the last two years, for investing many hours, from overseas, in making sure that I do things right. I would like to express my gratitude to Kedar Namjoshi for giving me the opportunity to experience research at Bell Labs, and for continuing with a fruitful collaboration afterwards. Finally, thanks for all the constructive comments and advice that I received from my other committee members: Benjamin Goldberg, Clark Barrett and Zvi Kedem, as well as from Patrick Cousot who also supported me with his grants.

Above all I would like to thank my family; My wife Keren and my parents who provided all the love and support that I needed in this very long, and often quite tough, journey. I could not have done this otherwise.

Abstract

The work reported here focuses on two problems, that of synthesizing systems from formal specifications, and that of formalizing REST – a popular web applications' development pattern.

For the synthesis problem, we distinguish between the synchronous and the asynchronous case. For the former, we solve a problem concerning a fundamental flaw in specification construction in previous work. We continue with exploring effective synthesis of asynchronous systems (programs on multi-threaded systems). Two alternative models of asynchrony are presented, and shown to be equally expressive for the purpose of synthesis.

REST is a software architectural style used for the design of highly scalable web applications. Interest in REST has grown rapidly over the past decade. However, there is also considerable confusion surrounding REST: many examples of supposedly RESTful APIs violate key REST constraints. We show that the constraints of REST and of RESTful HTTP can be precisely formulated within temporal logic. This leads to methods for model checking and run-time verification of RESTful behavior. We formulate several relevant verification questions and analyze their complexity.

Table of Contents

Dedication v

Acknowledgments vi

Abstract vii

List of Figures x

1 Introduction **1**

2 Revisiting Synthesis of GR(1) Specifications **5**

2.1 Introduction 5

2.2 Preliminaries 10

2.2.1 Temporal Logic and Tree-Models 10

2.2.2 Realizability of Temporal Specifications 13

2.3 The Syntactic Reduction is Incomplete 14

2.3.1 Synthesis of Reactive(1) Designs 15

2.3.2 Incompleteness 17

2.4 Well-Separated Environments 22

2.5 General Specifications 28

2.6 On the Connection Between Games and Trees 32

2.6.1 Game Structures 33

2.6.2 Realizability in Game Structures and in Tree Models 36

2.7 Conclusions 39

3 Effective Synthesis of Asynchronous Systems from GR(1) Specifications **40**

3.1 Introduction 40

3.2 Preliminaries 44

3.2.1 Temporal Logic 44

3.2.2 Realizability of Temporal Specifications 47

3.2.3 Structure and Notations of Specifications 51

3.2.4 The Rosner Reduction 52

3.3 Expanding the Rosner Reduction to Multiple Variables 54
3.4 A More General Asynchronous Interaction Model 65
3.4.1 A General (Multi-Core) Model 65
3.4.2 A Modified Generalized Rosner Reduction 71
3.5 Proving Unrealizability of a Specification 77
3.5.1 Over-Approximating the Kernel Formula 77
3.5.2 Applying the Unrealizability Test 81
3.6 Proving Realizability of a Specification, and Synthesis 84
3.6.1 Under-Approximating the Kernel Formula 85
3.6.2 Using the Under-Approximation, and synthesis 91
3.6.3 Applying the Realizability Test 103
3.6.4 A Possible Direction for Handling More Specifications 108
3.7 Conclusions and Future Work 110

4 Formalization and Automated Verification of RESTful Behavior 113
4.1 Introduction 113
4.2 REST and its Formalization 115
4.2.1 Building Blocks for REST 115
4.2.2 Formalizing Resource-Based Applications 117
4.2.3 Formalization of RESTful Behavior 119
4.3 REST on HTTP, and Variations 127
4.3.1 A Formal HTTP Model 127
4.3.2 RESTful HTTP Properties 129
4.3.3 Variations on RESTful HTTP Properties 135
4.3.4 Distinguishing REST from HTTP 140
4.4 Automated Verification of RESTful Behavior 141
4.4.1 Computation Model 142
4.4.2 Fundamental Questions 142
4.4.3 Automata Constructions 143
4.4.4 Model-Checking for Fixed Instances 144
4.4.5 Parameterized verification 149
4.4.6 Run-Time Monitoring 150
4.4.7 Synthesizing Servers 151
4.4.8 Relaxing The Atomicity of Communications 153
4.5 Related Work and Conclusions 154

5 Summary 156

Bibliography 160

List of Figures

3.1 The temporal hierarchy of properties 64
3.2 Algorithm for extracting T_a from T_s 94
3.3 The 'real' (R) and T_s (L) computations 97
3.4 ILTS (as an automaton) 105

Chapter 1

Introduction

At the heart of the field of formal methods is the notion of *formalization*. By formalizing reactive system's behavior one could describe both what is desired behavior, as well as what is undesired behavior. Such formal descriptions, expressed in a mathematical language, could then be used for making exact, proven, statements about systems' properties.

The two most common forms of such formalizations are the *temporal logics* Linear Temporal Logic (LTL) [Pnu77], and Computation Tree Logic (CTL) [EC80]. Both temporal logics are extensions of classical mathematical logic, which expresses relations between values of variables, to account for the dimension of time. Therefore, these logics express relations between values of variables, over infinite sequences of assignments to these variables. While classical mathematical logic could express notions such as "if a then b", temporal logics allow for the formalization of notions such as "always a" or even "eventually, a or b". While LTL considers linear time and single temporal computations, and therefore could express statements of the form "a since b", CTL considers the set of all 'possible' futures and could express statements such as "there exists some future where always a". Nei-

ther one of these two temporal logics can express all properties that the other can. There are other forms of formalizations (e.g., Live Sequence Charts [DH99]), but we shall not discuss them in this dissertation.

Using a formal behavioral description, in the form of a *temporal formula* in the case of temporal logics, one could pursue either one of two possible directions. The first one is that of *verification*, in which an existing system's properties are analyzed. Such analysis would include automated, or semi-automated, methodologies for proving relationships between possible behaviors of said system, and the behaviors expressed by the temporal formula. For example, one may wish to verify whether the considered system is guaranteed to never reach some critical state, or that it always follows some assumption. The most common automated methodology is perhaps *model checking* [CGP99], which is used for verifying the behavior of systems.

The second direction that could be pursued given a formalization of a desired system behavior, is that of *synthesis*. Synthesis is, in some sense, an attempt to identify systems that are compliant with formalizations in a direction opposite to that of verification. While verification requires as input a system which is then tested for compliance with a behavior expressed by a temporal formula, synthesis is the attempt to automatically form a system that is guaranteed, **by construction**, to comply with said desired behavior. It is important to note that while for the purpose of verification temporal formulae may include variables that refer to 'internal states' of the system that are strongly connected to its implementation (e.g., the value of some internal counter), when used for the purpose of synthesis such formulae are expected to present a 'black box' representation of a desired system, describing only relationships between its *inputs* and *outputs*. Temporal

formulae that described desired systems for synthesis are called *specifications*.

Here, we use the formalizations of LTL and CTL to tackle both verification and synthesis. Chapter 2 and Chapter 3 concern with synthesis, while Chapter 4 concern with formalizing some behaviors for verification.

In Chapter 2 we revisit an approach to the synthesis of *synchronous systems* (e.g., hardware designs that respond to one clock) from LTL specifications that was presented first in [PPS06], and that was widely adopted since then as a basis for many extensions and applications. We describe and analyze cases in which that work produces incorrect results, and suggest how to correct it. As a result, the corrected algorithm provides a sound and complete solution for effective synthesis of a very expressive sub-class of LTL which is called $GR(1)$ formulae.

In Chapter 3 we actually use this corrected algorithm as one component in a presented methodology for the synthesis of *asynchronous systems* (e.g., a software thread in a multi-threaded environment that uses a scheduler as communications via shared memory) from LTL specifications. We first describe two models of asynchronous behavior, and then we use them to effectively reduce the problem of synthesizing asynchronous systems to that of synthesizing synchronous systems (both with multiple variables). The result is a sound and complete solution to the problem of synthesizing multiple-variable asynchronous systems from LTL. After discussing the high complexity of this theoretical solution, we follow with two simplifications of it that produce a sound yet incomplete solution to the problem of doing so effectively.

In Chapter 4 we develop a formalization – in LTL and CTL– of a concept that was earlier presented only *informally*, i.e., in a natural language. This concept is called REST, and is a design approach for clients-server systems that strives

to optimize scalability, robustness and modularity of such systems. We follow by presenting several ways in which our formalization could be used for the purpose of verification of such systems.

Finally, in Chapter 5, we conclude this dissertation.

Chapter 2

Revisiting Synthesis of GR(1) Specifications

2.1 Introduction

One of the most ambitious and challenging problems in reactive systems construction is the automatic synthesis of programs and (digital) designs from logical specifications. First identified as Church's problem [Chu63], several methods have been proposed for its solution ([BL69], [Rab72]). The two prevalent approaches to solving the synthesis problem are by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-player game. In these preliminary studies of the problem, the logical specification that the synthesized system should satisfy was given as an S1S formula.

This problem has been considered again in [PR89a] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts ([CE81], [MW84]) to synthesize closed programs from temporal specification, which reduced the synthesis problem to satis-

fiability. In order to synthesize open programs, the environment should be treated as an adversary – a case that is not reducible to satisfiability. The method proposed in [PR89a] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then determinized into a deterministic Rabin automaton $\mathcal{R}_\varphi$. This double translation may reach complexity of double exponent in the size of φ. Once the Rabin automaton is obtained, the game can be solved in time $n^{O(k)}$, where n is the number of states of the automaton (double exponential in the size of φ) and k is the number of accepting pairs (exponential in the size of φ). In fact, [PR89a] established a doubly exponential lower bound for the synthesis problem as well as for the realizability problem. The realizability problem – whether the specification is realizable – is to decide, for a given temporal specification, whether there exists a system that implements it. A negative answer to this question also rules out the manual construction of an implementation, and is not restricted to automatic construction methods.

The high complexity established in [PR89a] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Representative cases are the work in [AMPS98] which presents (besides the generalization to real time) efficient polynomial solutions ($O(N^2)$, where N is the size of the state space of the design, which is exponential in the size of the underlying specification) to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulae $\Box\, p$, $\Diamond\, p$, $\Box \Diamond\, p$, or $\Diamond \Box\, p$. A more recent paper is [AT04]

which presents efficient synthesis approaches for the LTL fragment consisting of a Boolean combinations of formulae of the form $\Box\, p$.

The results of [AMPS98] and [AT04] were generalized in [PPS06] into the wider class of *generalized Reactivity*(1) formulae (GR(1)), i.e., formulae of the form

$$(\Box \Diamond p_1 \wedge \cdots \wedge \Box \Diamond p_m) \ \rightarrow \ (\Box \Diamond q_1 \wedge \cdots \wedge \Box \Diamond q_n). \tag{2.1}$$

Following the developments in [KPP05], [PPS06] shows how synthesis problems whose specification is a GR(1) formula can be solved in time $O(N^3)$. Furthermore, the paper presents a (symbolic) algorithm for extracting a design (program) which implements the specification.

It is then suggested that solving games with GR(1) winning conditions can be used for synthesis of more general specifications. Particularly, [PPS06] suggests to handle synthesis of LTL specifications of the form

$$(I_e \wedge \Box\, S_e \wedge \textstyle\bigwedge_i \Box \Diamond L_e^i) \rightarrow (I_s \wedge \Box\, S_s \wedge \textstyle\bigwedge_j \Box \Diamond L_s^j), \tag{2.2}$$

where I_e, I_s, L_e^i, and L_s^j are state assertions and S_e and S_s characterize the transitions of the environment and the system, respectively. Specifications of the form in Formula 2.2 are then converted to a GR(1) game with the winning condition as in Formula 2.1. We refer to this conversion as ‘the syntactic reduction’. The advantages of the syntactic reduction is that it leads to a simpler GR(1) game that is more likely to be solved in practice. Also presented is an argument that the class of formulae of the form in Formula 2.2 is sufficiently expressive to provide complete specifications of many designs.

The work in [PPS06] has been extensively used. Its feasibility was demon-

strated in [BGJ+07a, BGJ+07b], which consider a design for the *Advanced High-Performance bus* (AHB) that is part of the ARM's *Advanced Micro-controller Bus Architecture* (AMBA) [Ltd99] and a generalized buffer. These case studies yielded an automaton that was then converted to a circuit that has been implemented and tested successfully. This, in fact, is the first time that real-life blocks have been automatically synthesized from their high-level temporal specifications. Further applications include usage in the context of production of robot controllers [CKGC+07, KGFP07b, KGFP07a, WTM09, WTM10b, WTM10a] and user programming [KPP09, KS09]. So in many interesting cases, specifications of reactive systems are in the subset of LTL handled by this technique. In such cases, in spite of the extremely high theoretical lower bound, synthesis of reactive systems from temporal specifications is practically feasible.

The algorithm in [PPS06] has also been extended in various ways. We mention the work of [SSR08, SS09] that attempts to extend the fragment of LTL handled by such techniques while still maintaining the algorithmic advantages. In [PK09], it is used to synthesize asynchronous systems (such as software that copes with a potentially adversarial scheduler).

Here we show that, in fact, the syntactic reduction from specifications as in Formula 2.2 to GR(1) games checks the realizability of formulae of the format

$$(I_e \rightarrow I_s) \wedge (I_e \rightarrow \Box(\boxminus S_e \rightarrow S_s)) \wedge ((I_e \wedge \Box S_e) \rightarrow (\bigwedge_i \Box \Diamond L_e^i \rightarrow \bigwedge_j \Box \Diamond L_s^j)). \quad (2.3)$$

Formula 2.3 highlights the system's obligation not to be the first to violate safety. We prove that in cases that Formula 2.3 is realizable, then the design that realizes it also realizes Formula 2.2 (with the same components). However, we give a counter example, due to Roveri et al. [RBTJ06], to the other direction. Thus, the

syntactic reduction is not complete and may produce false negatives. In some cases an implication as in Formula 2.2 is realizable but the use of the syntactic reduction will declare that it is not.

We identify a condition on the specification of the environment that enables to use the syntactic reduction without losing completeness. This condition, which we call 'well separation', calls for environments that cannot be forced to violate their specifications.[1] Effectively, this condition can be checked using the solution of GR(1) games. In order to check whether the system can force the environment to violate its specification, a modified GR(1) game is created. In this game, the goal of the system is to avoid infinite computations that satisfy the environment's specification. If the system cannot do that, then the environment is well separated. We show that for well-separated environments the syntactic reduction is both sound and complete (for Formula 2.2). That is, if for a well-separated environment the syntactic reduction leads us to declare that the specification is unrealizable then this is indeed the case.

Finally, we consider the general case of specifications that are not well separated. We show that these types of specifications can be transformed so that the syntactic reduction can be applied on them. The transformation includes effectively a reduction of the system's safety requirements to a liveness requirement. The added power of the system, to violate its safety sacrificing its ability to satisfy liveness, pays off only in cases where the system can force the environment to falsify its own requirements. So finally, even for general specifications, the efficient syntactic reduction can be used with only a small increase in complexity.

To summarize, our contributions are the following.

[1] The notion of well-separation is related to *closure of specifications* [AL91].

1. Identify the incompleteness of the syntactic reduction.

2. Suggest a condition that is checkable with the same realizability techniques and that ensures that the syntactic reduction is complete.

3. Offer a transformation of general specifications that enables to use the syntactic reduction.

The rest of the chapter is organized as follows. In Section 2.2 we cover the basic definitions about temporal logic, models, and realizability. In Section 2.3 we show that the syntactic reduction is incomplete. In Section 2.4 we introduce the 'well separation' condition and in Section 2.5 we show how to safely use the syntactic reduction. We conclude in Section 2.7.

This chapter is based on published work; Most of what presented here was published as a conference paper [KP11].

2.2 Preliminaries

2.2.1 Temporal Logic and Tree-Models

We describe the syntax and semantics of a general branching-time temporal logic. This logic is an extension of CTL* ([CES86, EH86, ES84, HT87]), obtained by admitting quantification over propositional variables.

Let AP be a set of Boolean variables. The syntax of the logic is defined according to the following grammar.

$$\begin{aligned} \alpha &::= p \parallel \neg\alpha \parallel \alpha \vee \alpha \parallel (\exists p)\alpha \parallel \mathbf{E}\varphi \\ \varphi &::= \alpha \parallel \neg\varphi \parallel \varphi \vee \varphi \parallel \bigcirc \varphi \parallel \ominus \varphi \parallel \boxminus \varphi \parallel \varphi \mathcal{U} \varphi \end{aligned}$$

where $p \in AP$ is a variable, α are state formulae, and φ are path formulae.

We use the following standard abbreviations: T for $p \vee \neg p$, F for $\neg\mathrm{T}$, $a \wedge b$ for $\neg(\neg a \vee \neg b)$, $a \rightarrow b$ for $\neg a \vee b$, $a = b$ for $(a \rightarrow b) \wedge (b \rightarrow a)$, $(\forall p)\alpha$ for $\neg(\exists p)(\neg\alpha)$, $\Diamond a$ for $\mathrm{T}\mathcal{U}a$, $\Box a$ for $\neg\Diamond\neg a$, $a\mathcal{W}b$ for $a\mathcal{U}b \vee \Box a$ and $\mathbf{A}\varphi$ for $\neg\mathbf{E}(\neg\varphi)$. For a set $X = \{x_1, \ldots, x_k\} \subseteq AP$ of variables, we write $(\exists X)\alpha$ for $(\exists x_1)\cdots(\exists x_k)\alpha$ and similarly for $(\forall X)\alpha$.

The logic LTL is obtained by considering only path formulae and disallowing the usage of the $\mathbf{E}$ and $\exists$ operators. In this chapter an LTL formula φ can be identified with the CTL* formula $\mathbf{A}\varphi$.

The semantics of temporal logic is given with respect to *models* of the form $M = \langle S, R, L, s_0\rangle$, where S is a countable set of *states*, $R \subseteq S \times S$ is a total *transition relation*, $s_0 \in S$ is an *initial state*, and L is the *labeling function*, assigning to each state $s \in S$ an *interpretation* $L(s) \subseteq AP$ of all Boolean variables true in s. A *path* in M is a maximal sequence $\pi = (s_0, s_1, \ldots)$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. For a variable $p \in AP$, we say that a model $M' = \langle S, R, L', s_0\rangle$ is a *p-variant* of M if for every $s \in S$ and every $q \neq p$ we have $q \in L(s)$ **iff** $q \in L'(s)$.

A model $M = \langle S, R, L, s_0\rangle$ is called a *tree-model*, if the following conditions are satisfied:

1. The initial state s_0, called the *root* of M, is the unique state in S which has no *parent*, i.e., no state $s \in S$, such that $R(s, s_0)$.

2. Every other state $t \neq s_0$, has precisely one parent.

3. For every state $s \in S$, there exists a unique path leading from s_0 to s.

Satisfiability of state formulae is defined with respect to a model M and a state s as follows:

1. $\langle M, s\rangle \models p$ for a variable $p \in AP$ **iff** $L(s)$ assigns T to p.

2. $\langle M, s\rangle \models \neg\alpha$ **iff** $\langle M, s\rangle \not\models \alpha$.

3. $\langle M, s\rangle \models \alpha \vee \beta$ **iff** $\langle M, s\rangle \models \alpha$ or $\langle M, s\rangle \models \beta$.

4. For a variable p, $\langle M, s\rangle \models (\exists p)\alpha$ **iff** $\langle M', s\rangle \models p$ for some M' which is a p-variant of M.[2]

5. For a path formula φ, $\langle M, s\rangle \models \mathbf{E}\varphi$ **iff** for some infinite path π in M starting in s, $\langle M, \pi\rangle \models \varphi$.

Satisfiability of a path formula is defined with respect to a model M, an infinite path π in M, and a location $i \in \mathbb{N}$ in π according to the following:

1. For $\pi = (s_0, \ldots)$ and a state formula α, $\langle M, \pi, i\rangle \models \alpha$ **iff** $\langle M, s_i\rangle \models \alpha$.

2. $\langle M, \pi, i\rangle \models \neg\varphi$ **iff** $\langle M, \pi, i\rangle \not\models \varphi$.

3. $\langle M, \pi, i\rangle \models \varphi \vee \psi$ **iff** $\langle M, \pi, i\rangle \models \varphi$ or $\langle M, \pi, i\rangle \models \psi$.

4. $\langle M, \pi, i\rangle \models \bigcirc\, \varphi$ **iff** $\langle M, \pi, i+1\rangle \models \varphi$.

5. $\langle M, \pi, i\rangle \models \ominus\, \varphi$ **iff** $i > 0$ and $\langle M, \pi, i-1\rangle \models \varphi$.

6. $\langle M, \pi, i\rangle \models \boxminus\, \varphi$ **iff** for all $j \leq i$ we have $\langle M, \pi, j\rangle \models \varphi$.

7. $\langle M, \pi, i\rangle \models \varphi \mathcal{U} \psi$ **iff** for some $j \geq i$, $\langle M, \pi, j\rangle \models \psi$, and for all k, $i \leq k < j$, $\langle M, \pi, k\rangle \models \varphi$.

We say that the infinite path $\pi = s_0, s_1, \ldots$ in model M satisfies the path formula φ, **iff** $\langle M, \pi, 0\rangle \models \varphi$.

We say that the model M satisfies the state formula α, and write $M \models \alpha$, **iff** $\langle M, s_0\rangle \models \alpha$. A state formula α is said to be *satisfiable* **iff** for some model M,

[2] Our definition restricts the value of p in a p-variant to be fixed according to the state of M. We use p-variants in the context of infinite trees where this is not a restriction.

$M \models \alpha$. The formula α is *valid*, denoted by $\models \alpha$, **iff** it is satisfied by every model. In case all variables in the state formula α are quantified, the labeling of M is not important and we may omit it.

2.2.2 Realizability of Temporal Specifications

Let X and Y be a partition of AP. A (semantic) *synchronous program* P from X to Y is a function $f_P : (2^X)^+ \mapsto 2^Y$. The intended meaning of this function is that it represents a program with a set of *Boolean inputs* X, and a set of *Boolean outputs* Y, such that at each step of the computation $i = 0, 1, \ldots$, the program outputs (assigns to Y) the value $f_P(a_0, a_1, \ldots, a_i)$, where $a_0, a_1, \ldots, a_i$ is the sequence of input values assumed by the variables of X over steps $0, 1, \ldots, i$.

A *full-X-tree* is a tree-model $M = \langle S, R, L, \epsilon \rangle$, where $S = (2^X)^*$, $R = \{(\pi, \pi \cdot a) \mid \pi \in (2^X)^* \text{ and } a \subseteq X\}$, and $L(\pi \cdot a) \cap X = a$. Thus, a full-$X$-tree is a structure whose states are named after *strings* of the elements of 2^X. The intuition behind a full-X-tree is that it should contain all the possible sequences of values of the variables of X. Given a full-X-tree M, we can interpret it as a program P^M, represented by the function f_{P^M}, such that

$$f_{P^M}(a_0, a_1, \ldots, a_i) = L(a_0 \cdot a_1 \cdot \ldots \cdot a_i) \cap Y. \tag{2.4}$$

Dually, a program f_P gives rise to the full-X-tree $M_P = \langle (2^X)^*, R_P, L_P, \epsilon \rangle$, where for every $\pi \in (2^X)^*$ and $a \subseteq X$ we have $L_P(\pi \cdot a) = \{a\} \cup f_P(\pi \cdot a)$ and $L_P(\epsilon) = \emptyset$. We say that a program f_P satisfies an LTL formula φ if the model $M_P \models \mathbf{A} \bigcirc \varphi$. Notice that ϵ is used as a 'dummy' state that collects all possible initial values of the program in one tree.

Definition 2.1 (realizability)**.** *An* LTL *specification* $\varphi(X, Y)$ *is* **realizable** *if there exists a program* P *that satisfies* $\varphi(X, Y)$*. Such a program* P *is said to be realizing* $\varphi(X, Y)$*.*

The following theorem is proven in [PR89a].

Theorem 2.1 ([PR89a])**.** *The following conditions are equivalent:*

1. *The specification* $\varphi(X, Y)$ *is realizable.*
2. *The formula* $(\forall X)(\exists Y)\mathbf{A}\varphi(X, Y)$ *is valid.*
3. *The formula* $\mathbf{A} \bigcirc \varphi(X, Y)$ *is satisfied by some full-*X*-tree.*

2.3 The Syntactic Reduction is Incomplete

In [PPS06], Piterman et al. suggest an approach for synthesis of "Reactive(1) designs". They show how to solve a *game structure* (for a definition, see Subsection 2.6.1) whose winning condition is of the form of Formula 2.1 (called GR(1) games) and how to extract a program that realizes GR(1) winning conditions. In this chapter, we use a different, yet equivalent, set of definitions. We describe this procedure as checking for the existence of tree-models of a specific family, called safe-X-trees (defined in Subsection 2.3.1), that satisfy a GR(1) formula. For short, we refer to this procedure as *checking for the existence of GR(1) trees*. Piterman et al. follow by describing a syntactic reduction from the problem of synthesis of formulae in the form of Formula 2.2 to checking for the existence of GR(1) trees (in their terminology, solving GR(1) games). For short, we refer to this reduction as *the syntactic reduction*. Here, we identify that the syntactic reduction introduces an error and falsefully declares some specifications as unrealizable. We present the syntactic reduction and identify the problems in it.

2.3.1 Synthesis of Reactive(1) Designs

Let X and Y be a partition of AP to input and output variables, respectively. We say that variables in X are *locally controlled* by the environment and variables in Y are *locally controlled* by the system. Consider a specification describing an interplay between a *system* s and an *environment* e. For every $\alpha \in \{e, s\}$, $\varphi_\alpha(X, Y)$ (which is the specification that defines the allowed actions of α) is a conjunction of:

1. I_α – a Boolean formula (equally, an assertion) over AP, describing the initial state of α. The formula I_s may refer to all variables and I_e may refer only to the variables X;

2. $\square S_\alpha$ (safety component) – a formula describing the transition relation of α, where S_α describes the update of the locally controlled state variables (identified by being *primed*, e.g., x' for $x \in X$) as related to the current state (unprimed, e.g., x), with the exception that s can observe X's next values;

3. L_α (liveness component) – each L_α is a conjunction of $\square \diamond p$ formulae where p is a Boolean formula.

In the case that a specification includes temporal past formulae instead of the Boolean formulae in any of the three conjuncts mentioned above, we assume that a pre-processing of the specification was done to translate it into another one that has the same structure but without the use of past formulae. This can be always achieved through the introduction of fresh Boolean variables that implement temporal testers for past formulae [PZ08]. Therefore, without loss of generality, we discuss in this work only such past-formulae-free specifications.

We abuse notations and write φ_α also as a triplet $\langle I_\alpha, S_\alpha, L_\alpha \rangle$.

Here we expose the syntactic reduction using the vocabulary defined in Section 2.2. The two approaches (games and trees) are equivalent, as it is well known that emptiness of tree automata and solution of two-player games are inter-reducible [Wil01]. We find that the exposition through trees makes it clear why the syntactic reduction is incomplete. For completeness of presentation we include in Subsection 2.6.2 a proof that the two approaches are the same.

Consider a pair of specifications $\varphi_\alpha(X, Y)$, for $\alpha \in \{e, s\}$, where $\varphi_\alpha = I_\alpha \wedge \square\, S_\alpha \wedge L_\alpha$. We define the formula

$$Imp(\varphi_e, \varphi_s) : (I_e \wedge \square\, S_e \wedge L_e) \rightarrow (I_s \wedge \square\, S_s \wedge L_s).$$

The *safe-X-tree* over φ_e and φ_s is a tree-model $M = \langle S, R, L, \epsilon \rangle$, where $S \subseteq (2^X)^*$ and $R \subseteq \{(\pi, \pi \cdot a) \mid \pi \in (2^X)^* \text{ and } a \subseteq X\}$. S, R and L are defined by induction as follows. As in full-X-trees, for every $\pi \cdot a \in (2^X)^+$ we have $L(\pi \cdot a) \cap X = a$. The root ϵ is in S and $L(\epsilon) = \emptyset$. For every $a \subseteq X$ such that $a \models I_e$ we have $a \in S$, $(\epsilon, a) \in R$, and $L(a) \models I_s$. That is, only the successors of the root ϵ that satisfy the initial condition of the environment are included in the tree. Furthermore, their label has to satisfy the initial condition of the system. For every $\pi \cdot a \in S$ and every $b \subseteq X$ such that $(L(\pi \cdot a), b') \models S_e$[3] we have $\pi \cdot a \cdot b \in S$, $(\pi \cdot a, \pi \cdot a \cdot b) \in R$, and furthermore $(L(\pi \cdot a), (L(\pi \cdot a \cdot b))') \models S_s$. That is, states in the tree have only the 'environmentally-safe' successors according to their labeling and their location in the tree. These 'environmentally-safe' successors must be labeled in a way that satisfies the safety of the system.

The work in [PPS06] describes an algorithm for checking the existence of a safe-X-tree (always, implicitly, over φ_e and φ_s) that satisfies specifications of the form

[3] b' is the primed version of b.

$\mathbf{A}(L_e \to L_s)$ (as in Formula 2.1). More accurately, the algorithm computes the set W of states that participate in some safe-X-tree and satisfy the specification $\mathbf{A}(L_e \to L_s)$. Then, if for every $a \subseteq X$ such that $a \models I_e$ there exists a state $\pi \in W$ such that $L(\pi) \cap X = a$ and $L(\pi) \models I_s$, the specification is declared realizable. The following is stated in [PPS06] without a proof.

Conjecture 2.1. *The specification $Imp(\varphi_e, \varphi_s)$ is realizable* **iff** *there is a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \to L_s)$.*

2.3.2 Incompleteness

Here we show that the implication in Conjecture 2.1 holds only in one direction.

Lemma 2.1. *If there is a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \to L_s)$ then $Imp(\varphi_e, \varphi_s)$ is realizable.*

Proof: Given a safe-X-tree over φ_e and φ_s M that satisfies $\mathbf{A}(L_e \to L_s)$ we construct a full-X-tree M' that agrees with the labeling of M on all states that appear in M and labels other states arbitrarily.

Consider an infinite path π in M'. If π appears also in M then π satisfies $\bigcirc I_e$, $\bigcirc I_s$, $\bigcirc \square S_e$, and $\bigcirc \square S_s$. By assumption, π also satisfies $L_e \to L_s$. It follows that π satisfies $Imp(\varphi_e, \varphi_s)$. If π does not appear also in M then either π does not satisfy $\bigcirc I_e$ or π does not satisfy $\bigcirc \square S_e$. Then π vacuously satisfies $Imp(\varphi_e, \varphi_s)$. It thus follows from Theorem 2.1 that $Imp(\varphi_e, \varphi_s)$ is realizable. $\square$

Roveri et al. [RBTJ06] gave the following counterexample to the other direction of Conjecture 2.1. Consider the following specification over input Boolean variable x and output Boolean variable y:

$$\varphi_1 : \quad \square \neg x' \wedge \square \diamond (x = y) \;\to\; \square (y' = x') \wedge \square \diamond y$$

The specification φ_1 is realizable. The program that maintains $y = \text{T}$ realizes it since it falsifies the left-hand side of the specification. Either the environment violates the requirement $\Box\neg x'$ and sets $x = \text{T}$ infinitely often, or the environment cannot fulfill its liveness requirement $\Box\Diamond(x = y)$. Therefore, the implication holds. On the other hand, there is no safe-X-tree that satisfies $\mathbf{A}(\Box\Diamond(x = y) \rightarrow \Box\Diamond y)$. Indeed, the safe-$X$-tree restricts the labels of nodes so that in every node (other than the root and its immediate successors) $y = x = \text{F}$. Thus, the only possible safe-X-tree has exactly four infinite paths. One path for each initial assignment for x and y, and then both x and y are always false. This tree does not satisfy the implication $\mathbf{A}(L_e \rightarrow L_s)$. The only way for the system to realize φ_1 is by violating its safety component, S_e. The search over safe-X-trees is too narrow, leading to the false proclamation that the specification is unrealizable.

Another counter example is the following:

$$\varphi_2 : \quad \neg x \wedge \Box(x' = x) \wedge \Box\Diamond(x \neq y) \;\rightarrow\; y \wedge \Box(y' = y) \wedge \Box\Diamond\neg y$$

Again, φ_2 is realizable. A possible implementation is a program that assigns to y the initial value F and falsifies I_s. For similar reasoning as in the previous case, the search over safe-X-trees by using the syntactic reduction is too narrow, declaring φ_2 to be unrealizable.

To address this problem, the authors of [PPS06] conjectured that checking for the existence of a safe-X-tree that satisfies $\mathbf{A}(L_e \rightarrow L_s)$ solves a different realizability problem (the modified formula was described in [Pnu06]). Consider two specifications $\varphi_\alpha(X, Y)$, for $\alpha \in \{e, s\}$, where $\varphi_\alpha = I_\alpha \wedge \Box S_\alpha \wedge L_\alpha$. We define

the formula:

$$Sep(\varphi_e, \varphi_s) : (I_e \to I_s) \wedge (I_e \to \Box(\boxminus S_e \to S_s)) \wedge ((I_e \wedge \Box S_e) \to (L_e \to L_s)).$$

Intuitively, this formula requires the system to match the co-operativeness level of the environment: if the environment satisfies its initial condition the system must do the same; as long as the environment does not violate its safety the system must not violate its safety; and if the environment satisfies all its requirements (initiallity, safety, and liveness) then the system must satisfy its liveness as well.[4] We formally prove the conjecture that $Sep(\varphi_e, \varphi_s)$ corresponds to the existence of safe-X-trees.

Theorem 2.2. *The following conditions are equivalent:*

1. *The specification* $Sep(\varphi_e, \varphi_s)$ *is realizable.*

2. *The formula* $\mathbf{A}(L_e \to L_s)$ *is satisfied by some safe-*X*-tree over* φ_e *and* φ_s*.*

Proof: We shall prove both directions:

$1 \Longrightarrow 2$: Since $Sep(\varphi_e, \varphi_s)$ is realizable, by Theorem 2.1, there exists a full-X-tree, $M = \langle S, R, L, \epsilon \rangle$, that satisfies the formula $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$. We now describe how to prune M into a safe-X-tree over φ_e and φ_s, $M' = \langle S', R', L', \epsilon \rangle$, that satisfies the formula $\mathbf{A}(L_e \to L_s)$. We define $S' \subseteq S$ and for every $s \in S'$, $L'(s) = L(s)$. For every $\{s, t\} \subseteq S'$ where $s \neq t$, $(s,t) \in R' \leftrightarrow (s,t) \in R$. The set S' is defined inductively, as follows. The root $\epsilon \in S$ is in S' and for every $a \subseteq X$ such that $a \models I_e$ we have $a \in S'$ (otherwise, naturally, $a \notin S'$). Since M is a full-X-tree, then for every such a, $a \in S$. For every $\pi \cdot a \in S'$

[4] A similar obligation, when only safety is involved, is called *strict realizability* in [BGHJ09].

and every $b \subseteq X$ such that $(L(\pi \cdot a), b') \models S_e$ we have $\pi \cdot a \cdot b \in S'$. Again, $\pi \cdot a \cdot b$ is otherwise excluded from S' and, again, since M is a full-X-tree, then for every such $\pi \cdot a \cdot b$, $\pi \cdot a \cdot b \in S$. We get that, indeed, $S \subseteq S'$.

Since M satisfies the formula $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$ and since M' is a subset of the paths of M, we get that also M' satisfies the formula $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$. Since M' must satisfy (as part of *Sep*) the formula $\mathbf{A} \bigcirc (I_e \to I_s)$, and since for every $a \in S'$ we have that $a \models I_e$, then for every $a \in S'$ we have that $L(a) \models I_s$. M' also satisfies the second conjunct in *Sep* - the formula $\mathbf{A} \bigcirc (I_e \to \Box(\boxminus S_e \to S_s))$. Since we know that it satisfies $\mathbf{A} \bigcirc I_e$, we get that M' satisfies $\mathbf{A} \bigcirc \Box(\boxminus S_e \to S_s)$. For every $(s,t) \in R'$, we have that $(L'(s), t') \models S_e$ and, finally, we get that M' satisfies $\mathbf{A} \bigcirc \Box S_s$ (which means that, for such (s,t), $(L'(s), (L'(t))') \models S_s$). We get that M' is a safe-X-tree over φ_e and φ_s.

As before, M' must satisfy the last conjunct of *Sep*, namely $\mathbf{A} \bigcirc ((I_e \wedge \Box S_e) \to (L_e \to L_s))$. Since M' is a safe-X-tree, this formula gives us that M' satisfies $\mathbf{A} \bigcirc (L_e \to L_s)$ which, due to the fact the the next operator ($\bigcirc$) has no impact on the satisfaction of liveness properties such as L_e and L_s, is logically equivalent to $\mathbf{A}(L_e \to L_s)$. Therefore, M' satisfies $\mathbf{A}(L_e \to L_s)$.

$2 \Longrightarrow 1$: Let $M = \langle S, R, L, \epsilon \rangle$ be a safe-X-tree over φ_e and φ_s that satisfies the formula $\mathbf{A}(L_e \to L_s)$. We describe how to complete it into a full-X-tree $M' = \langle S', R', L', \epsilon \rangle$ that satisfies the formula $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$. Since M' is a full-X-tree, then, clearly, $S \subseteq S'$. Similarly, $R \subseteq R'$ (both S' and R' are well defined by M' being a full-X-tree). We define $L'(s) = L(s)$ for every $s \in S$. For every $t \in S' \setminus S$ we set $L'(t) \cap Y$ to an arbitrary value.

By definition, M is a *sub-tree* of M', and all of the paths of M, together with their labels, are contained in M'. All such paths in M' (paths that also exist in M), are known to satisfy the formula $L_e \rightarrow L_s$. Therefore, they also satisfy the formula $\bigcirc(L_e \rightarrow L_s)$. These paths, as paths in a safe-X-tree over φ_e and φ_s, are also known to satisfy the formulae $\bigcirc I_e$, $\bigcirc I_s$, $\bigcirc \Box S_e$ and $\bigcirc \Box S_s$. Therefore, such paths must satisfy $\bigcirc\, Sep(\varphi_e, \varphi_s)$. It is left to show that paths in M' that are not paths in M also satisfy $\bigcirc\, Sep(\varphi_e, \varphi_s)$.

Let us consider such a path $\pi = (s_0, s_1, \ldots)$ in M'. Since it is not a path in a safe-X-tree over φ_e and φ_s, it must either falsify the formula $\bigcirc I_e$, or have a minimal index $i > 0$ for which $(L'(s_i), (L'(s_{i+1}))') \not\models S_e$ and π falsifies the formula $\bigcirc \Box S_e$. In the case that $\bigcirc I_e$ does not hold over π then, trivially, $\bigcirc\, Sep(\varphi_e, \varphi_s)$ does. In the second case, since we may assume that π satisfies the formula $\bigcirc I_e$, then, by the definition of a safe-X-tree over φ_e and φ_s, we get that $s_0, \ldots, s_i$ is a prefix of a path in M. Therefore, as L' is identical to L over the prefix of π, we conclude that π satisfies the formula $\bigcirc I_s$ and for all $1 \leq j < i$, $(L'(s_j), (L'(s_{j+1}))') \models S_e$. It follows that for all $1 \leq j < i$ we also have $(L'(s_j), (L'(s_{j+1}))') \models S_s$. Then, π satisfies the formula $\bigcirc \Box(\boxminus S_e \rightarrow S_s)$. Finally, since π falsifies the formula $\bigcirc \Box S_e$, the third conjunct in $Sep(\varphi_e, \varphi_s)$ also holds.

We get that for all the paths of M', the formula $\bigcirc\, Sep(\varphi_e, \varphi_s)$ holds. In other words, $\mathbf{A} \bigcirc\, Sep(\varphi_e, \varphi_s)$ is satisfied by M'. By Theorem 2.1 we get that *Sep* is realizable.

$\Box$

We emphasize that the the process for checking for the existence of GR(1) trees correctly solves the existence problem of a safe-X-tree that satisfies formulae of the form $\bigwedge_{i\in I} \Box \Diamond a_i \rightarrow \bigwedge_{j\in J} \Box \Diamond b_j$. However, the existence of such a safe-X-tree is equivalent to the realizability of the formula $\mathit{Sep}(\varphi_e, \varphi_s)$ and not to the realizability of $\mathit{Imp}(\varphi_e, \varphi_s)$. The following specifications, obtained by re-arranging the components of φ_1 and φ_2, are, unlike φ_1 and φ_2, unrealizable:

$$\varphi_1^{Sep}: \quad \Box(\boxminus \neg x' \rightarrow (y' = x')) \wedge (\Box \neg x' \rightarrow (\Box \Diamond (x = y) \rightarrow \Box \Diamond y))$$

$$\varphi_2^{Sep}: \quad (\neg x \rightarrow y) \wedge (\neg x \rightarrow \Box(\boxminus (x' = x) \rightarrow (y' = y))) \wedge$$
$$((\neg x \wedge \Box (x' = x)) \rightarrow (\Box \Diamond (x \neq y) \rightarrow \Box \Diamond \neg y))$$

2.4 Well-Separated Environments

We show now that, in some cases, the existence of a safe-X-tree does guarantee realizability of the implication between the specification for the environment and the specification for the system.

Given specifications $\varphi_\alpha(X, Y)$ for $\alpha \in \{e, s\}$, where $\varphi_\alpha = \langle I_\alpha, S_\alpha, L_\alpha \rangle$ consider the two formulae:

$$\mathit{Imp}(\varphi_e, \varphi_s) : (I_e \wedge \Box S_e \wedge L_e) \rightarrow (I_s \wedge \Box S_s \wedge L_s)$$

$$\mathit{Sep}(\varphi_e, \varphi_s) : (I_e \rightarrow I_s) \wedge (I_e \rightarrow \Box(\boxminus S_e \rightarrow S_s)) \wedge ((I_e \wedge \Box S_e) \rightarrow (L_e \rightarrow L_s))$$

Unfortunately, $\mathit{Sep}(\varphi_e, \varphi_s)$ is not a convenient formula for a specification writer. From the proof of Lemma 2.1 it is easy to deduce that $\mathit{Sep}(\varphi_e, \varphi_s) \rightarrow \mathit{Imp}(\varphi_e, \varphi_s)$. Clearly, the converse does not hold. The formula $\mathit{Imp}(\varphi_e, \varphi_s)$ seems more intuitive than $\mathit{Sep}(\varphi_e, \varphi_s)$ (as an assume-guarantee structure). It would be desirable to find a way to allow developers to hold on to it while using the efficient algorithm for checking for existence of GR(1) trees. We next identify cases for which the two

formulae are equi-realizable, and in which they are realized by the same programs.

Definition 2.2 (well separation)**.** *An environment e with a set of inputs X and a set of outputs Y, which is specified by $\varphi_e = \langle I_e, S_e, L_e \rangle$, is* **well separated** *if for every safe-X-tree M over φ_e and $\varphi_s = \langle \mathrm{T}, \mathrm{T}, \Box \Diamond \mathrm{F} \rangle$, every state in M participates in an infinite path π that satisfies $\bigcirc(I_e \wedge \Box S_e \wedge L_e)$.*

Intuitively, a well-separated environment is one which could always (from every reachable state and for every system) continue with an infinite computation that would satisfy all of its requirements. In other words, that no finite behavior of an arbitrary system would be able to force the environment to a point from which it cannot fulfill its own requirements. Systems interacting with a well separated environment, would have to either comply with all of their own requirements in order to satisfy the specification, or to force the environment to fail to comply with the environment's liveness.[5] This is checked by removing the system's initial and safety restrictions (by setting them to T) and adding an impossible liveness requirement ($\Box \Diamond \mathrm{F}$). Thus, in order to win, the system would have to force the environment to violate its own specification. In order for the environment to be well separated, no system should be able to do so. When dealing with well-separated environments, the assumption made in the syntactic reduction, which requires all initial states and safety components of a specification to be valid in every realizing program, is not restrictive.

Testing for Well Separation. With the specification $\varphi_s^{ws} = \langle \mathrm{T}, \mathrm{T}, \Box \Diamond \mathrm{F} \rangle$, the formula $Sep(\varphi_e, \varphi_s^{ws})$ becomes logically equivalent to $\neg(I_e \wedge \Box S_e \wedge L_e)$. Since

[5] The authors of [CHJ08] stress the importance of having realizable environment specifications. Well separation is stronger, as it requires the environment to win from every state in every safe-X-tree and not just from the root. Realizability of the environment is not strong enough to ensure equivalence of *Imp* and *Sep*.

well-separated environments are environments that could not be forced to violate their specification $I_e \wedge \square S_e \wedge L_e$, it follows that for well-separated environments the specification $Sep(\varphi_e, \varphi_s^{ws})$ is unrealizable. The algorithm that checks for existence of GR(1) trees identifies whether there exists a safe-X-tree that satisfies $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$. It does so, however, by computing the set W of states in safe-X-trees from which the system can realize the following LTL formula.

$$\square(\boxminus S_e \rightarrow S_s) \wedge ((\square S_e \wedge L_e) \rightarrow L_s) \tag{2.5}$$

The algorithm then checks whether for every $a \subseteq X$ such that $a \models I_e$ there exists a state $\pi \in W$ such that $L(\pi) \cap X = a$ and $L(\pi) \models I_s$. Formula 2.5, with φ_s^{ws}, becomes logically equivalent to $\neg(\square S_e \wedge L_e)$. Since for well-separated environments the system could never (from no state in a safe-X-tree) force the environment to violate $\square S_e \wedge L_e$, we get that not only is the specification $Sep(\varphi_e, \varphi_s^{ws})$ unrealizable for well-separated environments, but, in fact, in this case the set W constructed by the algorithm that checks for existence of GR(1) trees must be empty. Since $Sep(\varphi_e, \varphi_s^{ws})$ and $Imp(\varphi_e, \varphi_s^{ws})$ are logically equivalent, well separation of an environment could be tested efficiently using the syntactic reduction without worrying about a specification's syntax. This is done by testing for emptiness of the relevant set W that is constructed for $Imp(\varphi_e, \varphi_s^{ws})$. There is one exception to this procedure - in the case that $I_e = \text{F}$, the set W that is created while testing for well separation might still be empty despite the fact that such environments are, clearly, not well separated.

Theorem 2.3. *Let φ_e be the specification of a well-separated environment with a set of inputs X and a set of outputs Y. Then, for every specification φ_s and*

program P the following conditions are equivalent:

1. *The program P realizes $Imp(\varphi_e, \varphi_s)$.*

2. *The program P realizes $Sep(\varphi_e, \varphi_s)$.*

Proof: By Lemma 2.1, we have $2 \Longrightarrow 1$. We prove that $1 \Longrightarrow 2$. Let $\varphi_s = \langle I_s, S_s, L_s \rangle$ be some specification. Let $M = \langle S, R, L, \epsilon \rangle$ be the full-X-tree that corresponds to a program P that realizes $Imp(\varphi_e, \varphi_s)$ (as described in Subsection 2.2.2). Then, M satisfies the formula $\mathbf{A} \bigcirc Imp(\varphi_e, \varphi_s)$. We show that M also satisfies the formula $\mathbf{A} \bigcirc Sep(\varphi_e, \varphi_s)$.

Consider an infinite path $\pi = s_0, s_1, \ldots$ in M. By assumption, $\langle M, \pi, 1 \rangle \models Imp(\varphi_e, \varphi_s)$. It is either the case that π satisfies the left-hand-side of the implication or not.

- Consider the case that $\langle M, \pi, 1 \rangle \models \varphi_e$. Then, $\langle M, \pi, 1 \rangle \models \varphi_s$ as well. In this case, π satisfies every one of the conjuncts in $Sep(\varphi_e, \varphi_s)$ and it follows that $\langle M, \pi, 1 \rangle \models Sep(\varphi_e, \varphi_s)$.

- Consider the case that $\langle M, \pi, 1, \rangle \not\models \varphi_e$. Then one of the following three conditions holds.

  - If $\langle M, \pi, 1 \rangle \not\models I_e$. Then, as I_e appears on the left-hand-side of every implication in $Sep(\varphi_e, \varphi_s)$ we conclude that $\langle M, \pi, 1 \rangle \models Sep(\varphi_e, \varphi_s)$.

  - If $\langle M, \pi, 1 \rangle \models I_e$ but $\langle M, \pi, 1 \rangle \not\models \Box S_e$. Let $i > 0$ be the minimal location in π such that $(s_i, s'_{i+1}) \not\models S_e$. By the environment being well separated, the state s_i in M appears on an infinite path π' such that $\langle M, \pi', 1 \rangle \models I_e \wedge \Box S_e \wedge L_e$. By M being a program for $Imp(\varphi_e, \varphi_s)$ it must be the case that $\langle M, \pi', 1 \rangle \models \varphi_s$ as well. However, π and π'

share the prefix $s_0, \ldots, s_i$. It follows that for every $1 \leq j < i$ we have $(s_j, s'_{j+1}) \models S_s$ and $s_1 \models I_s$. Hence, $\langle M, \pi, 1\rangle \models I_e \rightarrow I_s$ and $\langle M, \pi, 1\rangle \models I_e \rightarrow \Box(\boxminus S_e \rightarrow S_s)$. As $\langle M, \pi, 1\rangle \not\models \Box S_e$, it follows that the last conjunct in $Sep(\varphi_e, \varphi_s)$ holds as well. Thus, $\langle M, \pi, 1\rangle \models Sep(\varphi_e, \varphi_s)$.

– If $\langle M, \pi, 1\rangle \models I_e \wedge \Box S_e$ but $\langle M, \pi, 1\rangle \not\models L_e$. By the environment being well separated, every state s_i in π participates in a path π_i such that $\langle M, \pi_i, 1\rangle \models \varphi_e$. As before, the paths π_i and π share their prefix. It follows, that it must be the case that $s_1 \models I_s$ and that for every $i \geq 1$ we have $(s_i, s'_{i+1}) \models S_s$. Thus, $\langle M, \pi, 1\rangle \models I_e \rightarrow I_s$ and $\langle M, \pi, 1\rangle \models I_e \rightarrow \Box(\boxminus S_e \rightarrow S_s)$. However, by assumption $\langle M, \pi, 1\rangle \not\models L_e$. It follows that $\langle M, \pi, 1\rangle \models (I_e \wedge \Box S_e \wedge L_e) \rightarrow L_s$ as the antecedent does not hold. We conclude that $\langle M, \pi, 1\rangle \models Sep(\varphi_e, \varphi_s)$.

$\Box$

One consequence of Theorem 2.3 is that with well-separated environments, $Imp(\varphi_e, \varphi_s)$ and $Sep(\varphi_e, \varphi_s)$ are equi-realizable and then there is no restriction on using the syntactic reduction, regardless of the syntax of the specification.

The specifications on which the synthesis algorithm is demonstrated in [PPS06] are realizable and produce, therefore, correct realizability and synthesis results. The specifications φ_1 and φ_2 from Section 2.3 contain, however, environments that are not well separated. Therefore, we cannot handle them, at least in their current form, using the algorithm at hand.

Theorem 2.3 mentions two sets of specifications: The set of specifications with well separated environments, and the set of specifications φ for which $Imp(\varphi_e, \varphi_s)$

and $Sep(\varphi_e, \varphi_s)$ are equi-realizable (by the same programs), which we call *syntactically equi-realizable*. While Theorem 2.3 proves that the former is a subset of the latter, the opposite is not true. Indeed, there exist specifications $Imp(\varphi_e, \varphi_s)$ with environments that are not well separated which are realizable, while there exists a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \rightarrow L_s)$. Such specifications fall in the gap between the set of specifications with well separated environments, and the set of syntactically equi-realizable specifications. One trivial example is the following specification over output Boolean variable y (without any input variable):

$$\varphi_3 : \quad \Box\, y \;\rightarrow\; \Box \Diamond\, y$$

Clearly, φ_3 contains an environment that is not well separated. Also, a safe-X-tree over $\varphi_{3,e} = \Box\, y$ and $\varphi_{3,s} = \Box \Diamond\, y$ that satisfies $\mathbf{A}(\mathrm{T} \rightarrow \Box \Diamond\, y)$ exists: That would be a tree that labels the variable y with the value $y = \mathrm{T}$ everywhere. For specifications φ for which $Imp(\varphi_e, \varphi_s)$ and $Sep(\varphi_e, \varphi_s)$ are equi-realizable but φ_e is not well separated, there exists a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \rightarrow L_s)$, realizing one strategy for the system. At the same time, the system has a (different) strategy that would cause the environment to violate its specification φ_e. In the case of φ_3, the first strategy would be to always assign $y = \mathrm{T}$. A second strategy would be one that, at some point, assigns $y = \mathrm{F}$; This strategy cannot be represented by a safe-X-tree over $\varphi_{3,e}$ and $\varphi_{3,s}$.

It seems reasonable to assume that most 'natural' specifications that developers may (or, perhaps more accurately, should) come up with would contain well-separated environments. The reason for this assumption is that usually a desired implementation of a specification would not be one in which the program tries to force the environment to become 'stuck' while ignoring its own (the program's)

requirements, but rather one that tries to fulfill them.

2.5 General Specifications

From the results in Section 2.4 we conclude that if the environment specification is well separated, we can use the syntactic reduction to determine realizability of the implication of the two specifications. In this section we show a transformation that allows to use the syntactic reduction for all specifications. Thus, specifications that are written in the intuitive-to-design form of $\mathit{Imp}(\varphi_e, \varphi_s)$ can be handled by the same methods.

Consider two specifications $\varphi_\alpha(X, Y)$, for $\alpha \in \{e, s\}$ as before. We introduce two fresh Boolean output variables t_I and t_S that implement temporal testers ([PZ08]) for $\boxminus(\circleddash \top \vee I_s)$ and for $\boxminus S_s$, respectively, and modify the specification φ_s as follows.

- $\widetilde{I_s} \quad = \quad (t_I = I_s) \wedge t_S$
- $\widetilde{S_s} \quad = \quad (t_I' = t_I) \wedge (t_S' = (t_S \wedge S_s))$
- $\widetilde{L_s} \quad = \quad \square \diamondsuit t_I \wedge \square \diamondsuit t_S \wedge L_s$

Denote $\widetilde{\varphi_s} = \langle \widetilde{I_s}, \widetilde{S_s}, \widetilde{L_s} \rangle$. Intuitively, we remove all restrictions on the way the system starts and on the way it updates its variables. However, the variables t_I and t_S memorize whether the system violated its initial condition or its safety requirement. Then, the liveness requirement of the system is augmented by a requirement to satisfy its initial condition and safety specification. Thus, the system may violate its initial condition or safety component (which is now a part of its liveness) in cases that it can force the environment to violate its own requirements. With this new form of the specification, every tree model that satisfies the environment's

initial condition and safety component, can be completed into a safe-X-tree over φ_e and $\widetilde{\varphi_s}$.

We now show that $Imp(\varphi_e, \varphi_s)$ and $Sep(\varphi_e, \widetilde{\varphi_s})$ are equi-realizable. Thus, by using the syntactic reduction on φ_e and $\widetilde{\varphi_s}$ gives the correct answer (and correct program) for $Imp(\varphi_e, \varphi_s)$. Formally, we have the following.

Theorem 2.4. *The following conditions are equivalent:*

1. *The specification $Imp(\varphi_e, \varphi_s)$ is realizable.*

2. *The specification $Sep(\varphi_e, \widetilde{\varphi_s})$ is realizable.*

Furthermore, every program that realizes the one realizes the other.

To satisfy the last clause of Theorem 2.4 a program that realizes $Imp(\varphi_e, \varphi_s)$ should be modified so that it also generates outputs for t_I and t_S, and vice versa.

Proof: We shall prove both directions:

$1 \Longrightarrow 2$: We show that a full-X-tree that realizes $Imp(\varphi_e, \varphi_s)$ could be augmented with the values of t_I and t_S such that it becomes a full-X-tree that realizes $Sep(\varphi_e, \varphi_s)$. Let $M = \langle S, R, L, \epsilon\rangle$ be a full-X-tree that realizes $Imp(\varphi_e, \varphi_s)$. We construct the full-$X$-tree $M' = \langle S, R, L', \epsilon\rangle$. For every $p \in X \cup Y$ and for every state s in S we have $p \in L(s)$ **iff** $p \in L'(s)$. Consider a state $a \subseteq X$ in S. If $L(a) \models I_s$ we set $t_I \in L'(a)$ and for every path $\pi \in (2^X)^+$ we set $t_I \in L'(a \cdot \pi)$. If $L(a) \not\models I_s$ we set $t_I \notin L'(a)$ and for every path $\pi \in (2^X)^+$ we set $t_I \notin L'(a \cdot \pi)$. We now add the label t_S by induction. For every state $a \subseteq X$ in S we set $t_S \in L'(a)$. Consider a state $a_0 \cdot \ldots \cdot a_n$ such that for all $0 \leq i \leq n$ we have $a_i \subseteq X$ and for every state $a_0 \cdot \ldots \cdot a_j$ for $j < n$ we have already set whether t_S is in $L'(a_0 \cdot \ldots \cdot a_j)$. If $t_S \in L'(a_0 \cdot \ldots \cdot a_{n-1})$ and

$(L(a_0 \cdot \ldots \cdot a_{n-1}), (L(a_0 \cdot \ldots \cdot a_n))') \models S_s$ then we set $t_S \in L'(a_0 \cdot \ldots \cdot a_n)$. Otherwise, we set $t_S \notin L'(a_0 \cdot \ldots \cdot a_n)$. By construction of M' we have that $\langle M', \epsilon \rangle \models \mathbf{A} \bigcirc(\widetilde{I_s} \wedge \Box \widetilde{S_s})$. All paths of M' must, therefore, trivially satisfy the first two conjuncts of $\bigcirc\, Sep(\varphi_e, \widetilde{\varphi_s})$.

We show now that all paths of M' satisfy the third conjunct of $\bigcirc\, Sep(\varphi_e, \widetilde{\varphi_s})$ as well. Consider an infinite path $\pi = a_0, \ldots$ in M'. By choice of M, we have $\langle M', \pi, 1 \rangle \models (\varphi_e \to \varphi_s)$.

- If $\langle M', \pi, 1 \rangle \models \varphi_e$ then, by assumption, $\langle M', \pi, 1 \rangle \models \varphi_s$. It follows that $L(a_1) \models I_s$. Hence, for every state s in π we have $t_I \in L'(s)$ and, in particular, $\langle M', \pi, 1 \rangle \models \Box \Diamond t_I$. It is also the case that for every $i > 0$ we have $(L(a_i), (L(a_{i+1}))') \models S_s$. Hence, for every state s in π we have $t_S \in L'(s)$ and in particular $\langle M', \pi, 1 \rangle \models \Box \Diamond t_S$. Since $\langle M', \pi, 1 \rangle \models L_s$, we finally get that $\langle M', \pi, 1 \rangle \models \widetilde{L_s}$ and the third conjunct of $\bigcirc\, Sep(\varphi_e, \widetilde{\varphi_s})$ holds.
- If $\langle M, \pi, 1 \rangle \not\models \varphi_e$, then the third conjunct of $\bigcirc\, Sep(\varphi_e, \widetilde{\varphi_s})$ holds vacuously: $\langle M, \pi, 1 \rangle \models (I_e \wedge \Box S_e) \to (L_e \to \widetilde{L_s})$ (since either I_e, $\Box S_e$, or L_e is false here).

It follows that π satisfies $\bigcirc\, Sep(\varphi_e, \widetilde{\varphi_s})$ and that, therefore, M' realizes $Sep(\varphi_e, \widetilde{\varphi_s})$.

Since the program that corresponds to M' could be easily extracted, step-by-step, from the program that corresponds to M (by evaluating the values of t_I and of t_S at each step based on their previous values), we say that, essentially, the program of M realizes both specifications.

$2 \Longrightarrow 1$: Consider a full-X-tree $M' = \langle S, R, L', \epsilon \rangle$ that realizes $Sep(\varphi_e, \widetilde{\varphi_s})$. Con-

sider the tree $M = \langle S, R, L, \epsilon \rangle$ that is obtained from M' by removing the information regarding t_I and t_S from the labeling of M'. We show that M realizes $Imp(\varphi_e, \varphi_s)$.

Consider a path π in M. If $\langle M, \pi, 1 \rangle \not\models \varphi_e$ then, trivially, $\langle M, \pi, 1 \rangle \models Imp(\varphi_e, \varphi_s)$. Consider the case that $\langle M, \pi, 1 \rangle \models \varphi_e$ and consider the same path π in M'. By assumption, $\langle M', \pi, 1 \rangle \models Sep(\varphi_e, \widetilde{\varphi_s})$. As $\langle M', \pi, 1 \rangle \models I_e$, it follows that $\langle M', \pi, 1 \rangle \models \widetilde{I}_s$. As $\langle M', \pi, 1 \rangle \models \Box\, S_e$, it follows that $\langle M', \pi, 1 \rangle \models \Box\, \widetilde{S}_s$. Finally, as $\langle M', \pi, 1 \rangle \models L_e$, it follows that $\langle M', \pi, 1 \rangle \models \Box \Diamond t_I \wedge \Box \Diamond t_S \wedge L_s$. As π satisfies $\Box \Diamond t_I$, it must be the case that $\langle M, \pi, 1 \rangle \models I_s$. Indeed, the value of I_s is equivalent to that of t_I, and t_I is either true for every state of the path or false for every state of the path. Similarly, as π satisfies $\Box \Diamond t_S$ it follows that t_S is true in every state of the path. Thus, it must be the case that $\langle M, \pi, 1 \rangle \models \Box\, S_s$. We conclude that $\langle M, \pi, 1 \rangle \models \varphi_s$ and that $\langle M, \pi, 1 \rangle \models Imp(\varphi_e, \varphi_s)$. Finally, M realizes $Imp(\varphi_e, \varphi_s)$, as required.

Since the program for M is the program for M' modified by dropping its assignments for t_I and for t_S, we could say that the two are the same.

□

Applying the above transformation to φ_1 and φ_2 from Section 2.3, we obtain

the following formulae:

$$\begin{array}{lll}
\widetilde{\varphi_1} & : & (\top \rightarrow ((t_I = \top) \wedge t_S)) \qquad \wedge \\
 & & (\top \rightarrow \square(\boxminus \neg x' \rightarrow ((t'_I = t_I) \wedge (t'_S = (t_S \wedge (y' = x')))))) \qquad \wedge \\
 & & ((\top \wedge \square \neg x') \rightarrow (\square \diamondsuit (x = y) \rightarrow (\square \diamondsuit t_I \wedge \square \diamondsuit t_S \wedge \square \diamondsuit y)) \\
\widetilde{\varphi_2} & : & (\neg x \rightarrow ((t_I = y) \wedge t_S)) \qquad \wedge \\
 & & (\neg x \rightarrow \square(\boxminus (x' = x) \rightarrow ((t'_I = t_I) \wedge (t'_S = (t_S \wedge (y' = y)))))) \qquad \wedge \\
 & & ((\neg x \wedge \square (x' = x)) \rightarrow (\square \diamondsuit (x \neq y) \rightarrow (\square \diamondsuit t_I \wedge \square \diamondsuit t_S \wedge \square \diamondsuit \neg y)))
\end{array}$$

By Theorem 2.4 we can use the syntactic reduction to prove that $\widetilde{\varphi_1}$ and $\widetilde{\varphi_2}$ are, indeed, realizable and the programs that realize them also realize φ_1 and φ_2.

One may ask why the approach advocated in this section should not be used always, avoiding the need to test environments for well separation. The simple answer is that the introduction of two new variables adds to the complexity of the synthesis process. Furthermore, part of the attractiveness of using the syntactic reduction is that including the system's safety as part of the requirements on the tree makes the search practically simpler.

2.6 On the Connection Between Games and Trees

In this chapter we define and prove everything in the language of tree models. The work in [PPS06], however, is defined using game structures. In this section we define game structures with two players - an environment and a system - and using definitions for game winning and for winning strategies, we prove that the two definitions are, in fact, interchangeable for our purposes.

2.6.1 Game Structures

We consider two-player games played between a system and an environment. The goal of the system is to satisfy a winning condition (derived, in the context of synthesis, from the specification) regardless of the actions of the environment. Formally, we have the following.

Let AP be a set of Boolean variables, and let X and Y be a partition of AP to input and output variables, respectively. A *game structure* $G = \langle V, \theta_e, \theta_s, \rho_e, \rho_s, \varphi\rangle$ consists of the following components:

- $V \subseteq 2^{AP}$ is a finite set of *states*.
- $\theta_e \subseteq 2^V$ is the *initial set of the environment*. For every $D \in \theta_e$, if $s \in D$, then for every $t \in V$ such that $t \cap X = s \cap X$, also $t \in D$. Also, for every two states $d_1, d_2 \in D$, $d_1 \cap X = d_2 \cap X$.
- $\theta_s \subseteq V$ is the *initial set of the system*.
- $\rho_e \subseteq V \times 2^V$ is the *transition relation of the environment*. If $(s, D) \in \rho_e$, then for every $t \in D$, and for every $u \in V$ such that $u \cap X = t \cap X$, also $u \in D$. Also, for every two states $d_1, d_2 \in D$, $d_1 \cap X = d_2 \cap X$.
- $\rho_s \subseteq V \times V$ is the *transition relation of the system*.
- φ is the *winning condition* for the system, given by an LTL formula.

For two states s and t of G, t is a *successor* of s if there exists a $D \subseteq V$ such that $t \in D$, $(s, D) \in \rho_e$, and $(s, t) \in \rho_s$. A *play* σ of G is a maximal sequence of states $\sigma : s_1, s_2, \ldots$ satisfying *initiality* namely $s_1 \in \theta_s \cap \bigcup \theta_s$, and *consecution* namely, for each $j \geq 1$, s_{j+1} is a successor of s_j. A play σ of a game structure G

progresses in the following way: from each state s in σ, the environment chooses a set of possible successors $D \subseteq V$ such that $(s, D) \in \rho_e$, and the system follows by choosing from this set a successor $t \in D$ that also agrees with its transition relation (i.e., $(s,t) \in \rho_s$). The first state in σ is chosen in a similar manner - first, the environment chooses a set of possible initial states $D \subseteq V$ such that $D \in \theta_e$, and the system follows by choosing from this set an initial state $s \in D$ that also agrees with its initial set (i.e., $s \in \theta_s$).

A play σ is *winning for the system* if it is infinite and satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is a partial function $f : V^* \times 2^V \mapsto V$ such that if $\sigma = s_0, \ldots, s_n$ is a finite prefix of a play, then for every $D \subseteq V$ such that $(s_n, D) \in \rho_e$ we have $(s_n, f(\sigma, D)) \in \rho_s$. Similarly, for every $D \subseteq V$ such that $D \in \theta_e$ we have $f(\lambda, D) \in \theta_s$ (λ here indicates an empty sequence of states). A play $s_1, s_2, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ and for all $D \in V$ such that $s_{i+1} \in D$ and $(s_i, D) \in \rho_e$ ($D \in \theta_e$ in the case that $i = 0$) we have $f((s_1, \ldots, s_i), D) = s_{i+1}$. Strategy f is *winning for the system* if all plays which are compliant with f are winning for the system. A game structure G is said to be *winning for the system* if there exists in G a winning strategy for the system. Dually, if a play, a strategy or a game structure is not winning for the system, it is said to be *winning for the environment*.

Given a specification $Imp(\varphi_e, \varphi_s)$, a set of input variables X and a set of output variables Y, the syntactic reduction involves constructing a game structure $G = \langle V, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ as follows. As described in Subsection 2.3.1, $\varphi_\alpha = \langle I_\alpha, S_\alpha, L_\alpha \rangle$ for $\alpha \in \{e, s\}$, where

- $V = 2^{X \cup Y}$. We denote states $s \in V$ as a pair $s = (s_X; s_Y)$, where $s_X \in 2^X$

and $s_Y \in 2^Y$.

- $\theta_e = \{\{(x; 2^Y)\} | x \in 2^X; x \models I_e\}$. That is, for a given $x \in 2^X$, $\{(x; 2^Y)\} = \{(x; y) | y \in 2^Y\}$.
- $\theta_s = \{(x; y) | x \in 2^X; y \in 2^Y; (x, y) \models I_s\}$.[6]
- $\rho_e = \{\{((x_1; y), (x_2; 2^Y))\} | x_1, x_2 \in 2^X; y \in 2^Y; (x_1, y, x_2') \models S_e\}$. That is, for given $x_1, x_2 \in 2^X$ and $y_1 \in 2^Y$,

  $\{((x_1; y_1), (x_2; 2^Y))\} = ((x_1; y_1), \{(x_2; y_2) | y_2 \in 2^Y\})$.
- $\rho_s = \{((x_1; y_1), (x_2; y_2)) | x_1, x_2 \in 2^X; y_1, y_2 \in 2^Y; (x_1, y_1, x_2', y_2') \models S_s\}$.
- $\varphi = (L_e \rightarrow L_s)$. That is, the winning condition is the implication between the liveness requirements of the two players.

To *solve* the game means to decide whether the game is winning for the system or for the environment. If the environment is winning the specification is declared *unrealizable*. Otherwise, if the system wins, it is declared *realizable*. In addition, if the specification is realizable, a winning strategy for the system is extracted.

In the next section we show how the above definitions are, in some sense, equivalent to the definitions for trees that were given in Section 2.3 and that are used in the main body of the chapter. This justifies our claims regarding

[6] In [PPS06], the authors defined the initial conditions, erroneously, united in a single element of the game structure that defined a set of all the states that satisfy $I_e \wedge I_s$. Such a definition (which assumes only a single possible initial state per game) fails to describe the correct manner in which plays develop, requiring the environment to make its first choice of an initial set of states and allowing the system to choose an initial state from that set. It also ignores the possibility that the system would win the game by the environment choosing an initial set that does not satisfy I_e. This mistake, however, was later discovered by Piterman who corrected and replaced it (in the authors' implementation of their algorithm) with an initial condition requirement that verifies that for every $x \in 2^X$ such that $x \models I_e$ there exists an initial state $(x; y)$ such that $(x, y) \models I_s$.

the work as it is presented in [PPS06]. There is a slight difference between our definitions and the definitions there, as we choose an *enumerative* representation of the games, i.e., states and transitions are declared explicitly, while [PPS06] choose a *symbolic* representation, i.e., states and transitions are declared implicitly through the variables and assertions over them.

2.6.2 Realizability in Game Structures and in Tree Models

Using the definitions from the previous section, we have the following.

Theorem 2.5. *The following conditions are equivalent:*

1. *The game structure constructed from the specification $Imp(\varphi_e, \varphi_s)$ is winning for the system.*
2. *There exists a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \rightarrow L_s)$.*

Proof: We shall prove both directions:

$1 \Longrightarrow 2$: Let f be a winning strategy for the system in the game structure $G = \langle V, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ that is constructed from $Imp(\varphi_e, \varphi_s)$. Such a strategy exists since G is winning for the system. We describe how to construct the safe-X-tree over φ_e and φ_s $M = \langle S, R, L, \epsilon \rangle$ and show that it satisfies the required formula.

We have $S \subseteq (2^X)^*$ that is defined inductively. $\epsilon \in S$ and $L(\epsilon) = \emptyset$. For every $a \subseteq X$ such that $a \models I_e$, we have $a \in S$ and $(\epsilon, a) \in R$. Since $\{(a; 2^Y)\} \in \theta_e$, $f(\lambda, \{(a; 2^Y)\})$ is defined and we set $L(a) = f(\lambda, \{(a; 2^Y)\})$. Since $f(\lambda, \{(a; 2^Y)\}) \in \theta_s$, we also know that $L(a) \models I_s$. For every $\pi \cdot a \in S$ and every $b \subseteq X$ such that $(L(\pi \cdot a), b') \models S_e$ we have $\pi \cdot a \cdot b \in S$

and $(\pi \cdot a, \pi \cdot a \cdot b) \in R$. Let $\pi = \pi_1, \ldots, \pi_n$ (it is possible that $n = 0$ and π is empty). Clearly, $f((L(\pi_1), L(\pi_1, \pi_2), \ldots, L(\pi), L(\pi \cdot a)), \{(b; 2^Y)\})$ is defined since $\{((a; L(\pi \cdot a)), (b; 2^Y))\} \in \rho_e$. Thus, we set $L(\pi \cdot a \cdot b) = f((L(\pi_1), L(\pi_1, \pi_2), \ldots, L(\pi), L(\pi \cdot a)), \{(b; 2^Y)\})$. Also, $(L(\pi \cdot a), (L(\pi \cdot a \cdot b))') \models S_s$ as $(L(\pi \cdot a), f((L(\pi_1), L(\pi_1, \pi_2), \ldots, L(\pi), L(\pi \cdot a)), \{(b; 2^Y)\})) \in \rho_s$. By construction, M is, indeed, a safe-X-tree over φ_e and φ_s. It is left to show that M satisfies $\mathbf{A}(L_e \to L_s)$.

By the construction of M, every infinite path $\pi = \pi_0, \pi_1, \pi_2, \ldots$ in M corresponds with a play $\sigma = L(\pi_1), L(\pi_2), \ldots$ of G that is compliant with f. Since f is a winning strategy, we know that σ satisfies $L_e \to L_s$ which gives us that $\langle M, \pi, 0 \rangle \models L_e \to L_s$. Finally, we get that M satisfies $\mathbf{A}(L_e \to L_s)$, as required.

$2 \Longrightarrow 1$: Let $M = \langle S, R, L, \epsilon \rangle$ be a safe-X-tree over φ_e and φ_s that satisfies $\mathbf{A}(L_e \to L_s)$. Based on M, we define f as a strategy for the system in the game structure $G = \langle V, \theta_e, \theta_s, \rho_e, \rho_s, \varphi \rangle$ that is constructed from $\mathit{Imp}(\varphi_e, \varphi_s)$. We then show that f is winning for the system.

The strategy f is defined inductively. For every $D \subseteq V$ such that $D \in \theta_e$ we choose some $d \in D$. We have that $d \cap X \models I_e$. That is, $d \cap X \in S$. We e set $f(\lambda, D) = L(d \cap X)$. This is unique definition since for every $d_1, d_2 \in D$, $d_1 \cap X = d_2 \cap X$. We know that for every $a \subseteq X$, where $a \in S$, we have $L(a) \models I_s$. Hence $f(\lambda, D) \models I_s$, and therefore we know that $f(\lambda, D) \in \theta_s$. For every finite prefix of a play $\sigma = s_0, \ldots, s_n$ and for every $D \subseteq V$ such that $(s_n, D) \in \rho_e$ we choose some $d \in D$. Inductively, we assume that $(s_0 \cap X) \cdot \ldots \cdot (s_n \cap X) \in S$. Clearly, for $n = 0$ this holds. We have that $(L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X)), (d \cap X)') \models S_e$. That is, $(s_0 \cap$

$X) \cdot \ldots \cdot (s_n \cap X) \cdot (d \cap X) \in S$. We set $f(\sigma, D) = L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X) \cdot (d \cap X))$. For the same reason as before, this definition is unique. We know that for every $a \subseteq X$, where $(s_0 \cap X) \cdot \ldots \cdot (s_n \cap X) \cdot a \in S$, we have $(L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X)), (L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X) \cdot a))') \models S_s$. Hence $(L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X)), (f(\sigma, D))') \models S_s$. Therefore we know that $(L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X)), f(\sigma, D)) \in \rho_s$. We conclude that f is, indeed, a strategy for the system in G.

For every play σ that is compliant with f the following holds. For every prefix $\sigma_n = s_0, \ldots, s_n$, we have $L((s_0 \cap X) \cdot \ldots \cdot (s_n \cap X)) = s_n$. Since M satisfies $\mathbf{A}(L_e \rightarrow L_s)$, every path in M satisfies this formula. In particular, the path $\epsilon, (s_0 \cap X), (s_0 \cap X) \cdot (s_1 \cap X), \ldots$ satisfies $L_e \rightarrow L_s$. That is, the computation $L(s_0 \cap X), L((s_0 \cap X) \cdot (s_1 \cap X)), \ldots$ satisfies $(L_e \rightarrow L_s)$. Finally, so does $s_0, s_1, \ldots = \sigma$. We conclude that f is a winning strategy for the system and that G is winning for the system.

$\square$

Theorem 2.5 justifies our usage of tree models instead of game structures when discussing the claims of [PPS06]. In particular, it proves that Conjecture 2.1 is interchangeable with the actual (and differently phrased) claim in that paper which states (without a proof):

Conjecture 2.2. *The specification Imp*(φ_e, φ_s) *is realizable* **iff** *the game structure constructed from it is winning for the system.*

2.7 Conclusions

Following the solution to LTL realizability suggested in [PPS06], we analyzed their syntactic reduction from LTL specifications to games. We showed when and how the syntactic reduction could be applied to LTL specifications to produce correct realizability results. We first presented a flaw in the syntactic reduction that may cause it to produce false-negatives. We then proved that the syntactic reduction solves the realizability of $Sep(\varphi_e, \varphi_s)$ rather than that of $Imp(\varphi_e, \varphi_s)$. We defined the class of well-separated environments for which the syntactic reduction produces correct realizability results and showed how to efficiently test for it. Finally, we proposed a transformation for systems' specifications, changing φ_s into $\widetilde{\varphi_s}$, that guarantees that the syntactic reduction would produce correct realizability results for all specifications without paying much in complexity.

One of the comments of users of this synthesis approach is that debugging unrealizable specifications is very problematic. We hope that highlighting the difference between $Sep(\varphi_e, \varphi_s)$ and $Imp(\varphi_e, \varphi_s)$ as well as the awareness to well-separated environments will alleviate some of the burden on users. In general, providing users of synthesis with helpful feedback in case of failure of synthesis as well as how to check the correctness of the specification are interesting issues for further research.

One question that remains open is how to identify specifications that are syntactically equi-realizable.

Chapter 3

Effective Synthesis of Asynchronous Systems from GR(1) Specifications

3.1 Introduction

One of the most ambitious and challenging problems in reactive systems design is the automatic synthesis of programs from logical specifications. It was suggested by Church [Chu63] and subsequently solved by two techniques [BL69, Rab72]. In [PR89a] the problem was set in a modern context of synthesis of reactive systems from Linear Temporal Logic (LTL) specifications. The synthesis algorithm converts a LTL specification to a Büchi automaton, which is then determinized [PR89a]. This double translation may be doubly exponential in the size of φ. Once the deterministic automaton is obtained, it is converted to a Rabin game that can be solved in time $n^{O(k)}$, where n is the number of states of the automaton (double exponential in φ) and k is a measure of topological complexity (exponential in φ). This algorithm is tight as the problem is 2EXPTIME-hard [PR89a].

This unfortunate situation led to extensive research on ways to bypass the

complexity of synthesis (e.g., [KV05, HP06, PPS06, PP06]). The works in [PPS06] is of particular interest to us. It achieves scalability by restricting the type of handled specifications. This led to many applications of synthesis in various fields [BGJ+07a, BGJ+07b, CKGC+07, KGFP07b, WTM09, WTM10b, WTM10a, KPP09, KS09, DBPU10, DBPU11]. These results can be summarized by stating that, in spite of the extremely high worst-case lower bound, synthesis of designs from their temporal specifications is feasible, due to the identification of restricted fragments of LTL which are adequate for expressing most design specifications and admit polynomial synthesis algorithms.

These results relate to the case of synchronous synthesis, where the synthesized system is synchronized with its environment. At every step, the environment generates new inputs and the system senses all of them and computes a response. This is the standard computational model for hardware designs.

Here, we are interested in synthesis of asynchronous systems. Namely, the system may not sense all the changes in the values of its inputs, and the responses computed by the system may become visible to the external world (including the environment) with an arbitrary delay. Furthermore, the system accesses one variable at a time while in the synchronous model all inputs are observed and all outputs are changed in a single step. This asynchronous model is the most appropriate for representing reactive software systems that communicate via shared variables on a multi-threaded platform.

We illustrate the difference between the two types of synthesis (synchronous and asynchronous) by the following trivial example. Consider a system with a single input x and a single output y, both Booleans. The behavioral specification

is given by the temporal formula

$$\varphi_1 : \Box(x \leftrightarrow y)$$

stating that, at all computation steps, it is required that the output should equal the input. Obviously, this specification calls for an implementation of a module that will consistently 'copy' the input to the output. As usual, the synthesis problem is to find a module such that, for all possible sequences of values appearing on input x, will maintain the specification φ_1.

It is not difficult to see that specification φ_1 is synchronously realizable. That is, there exists a synchronous module that maintains the specification φ_1. Such a module can be defined by having the initial condition $\theta : y \leftrightarrow x$ and the transition relation $\rho : y' \leftrightarrow x'$. This presentation is based on the notion of a Fair Discrete System (FDS) as presented, for example, in [PPS06]. (A hardware module implementing this specification would ultimately amount to a wire connecting the input to the output.)

On the other hand, the specification φ_1 is not asynchronously realizable. That is, there does not exist an asynchronous module such that, for all possible sequences of x-values, it will maintain the relation $x \leftrightarrow y$. The reason is that if x changes too rapidly, the system cannot observe all these changes and respond quickly enough. In particular, since in an asynchronous setting steps of the environment and of the system interleave, there is no way (unlike in the synchronous model) that x and y can both change in the same step.

In [PR89b], Pnueli and Rosner reduce asynchronous synthesis to synchronous synthesis. Their technique, which we call the Rosner reduction, converts a specification $\varphi(x; y)$ with single input x and single output y to a specification $\mathcal{X}(x, r; y)$.

The new specification relates to an additional variable r (r being a fresh Boolean input variable). They show that φ is asynchronously realizable **iff** $\mathcal{X}$ is synchronously realizable and how to translate a synchronous implementation of $\mathcal{X}$ to an asynchronous implementation of φ.

Our first result is an extension of the Rosner reduction to specifications with multiple input and output variables. Pnueli and Rosner assumed that the system alternates between reading its input and writing its output. For multiple variables, we assume cyclic access to variables: first reading all inputs, then writing all outputs (each in a fixed order). We show that this interaction mode is not restrictive as it is equivalent (with respect to synthesis) to the model in which the system chooses its next action (whether to read or to write, and which variable).

Combined with [PR89a], the reduction from asynchronous synthesis to synchronous synthesis presents a complete solution to the multiple-variables asynchronous synthesis problem. Unfortunately, much like in the synchronous case, it is not 'effective'. Furthermore, even if φ is relatively simple (for example, belongs to the class of $GR(1)$ formulae that is handled in [PPS06]), the formula $\mathcal{X}$ is considerably more complex and requires the full treatment of [PR89a].

Consequently, the methods presented in this paper propose various ways to approximate the expanded Rosner reduction without fully computing it. These methods are sound but not complete. In particular, we offer approximations from two sides as follows:

- Derive a weaker approximation $\mathcal{X}_{\downarrow}$ by removing a quantifier within $\mathcal{X}$. It can be shown that if $\mathcal{X}$ is synchronously realizable then so is $\mathcal{X}_{\downarrow}$. Equivalently, if $\mathcal{X}_{\downarrow}$ is synchronously unrealizable then so is $\mathcal{X}$. Furthermore, unlike $\mathcal{X}$, if φ belongs to the $GR(1)$ class then so does $\mathcal{X}_{\downarrow}$, and it could be effectively

analyzed using the work of [PPS06]. This shows that an effective way to check that φ is asynchronously unrealizable is to check that $\mathcal{X}_{\downarrow}$ is synchronously unrealizable.

- Derive a stronger approximation $\mathcal{X}_{\psi}$, such that for single input specifications, if $\mathcal{X}_{\psi}$ is synchronously realizable, then so is $\mathcal{X}$. We present a family of heuristics for finding $\mathcal{X}_{\psi}$ for a given specification φ, such that if φ belongs to the $GR(1)$ class then so does $\mathcal{X}_{\psi}$. This provides an effective way to check that φ is asynchronously realizable by checking that $\mathcal{X}_{\psi}$ is synchronously realizable. We also show how the synchronous synthesis of $\mathcal{X}_{\psi}$ could then be used to extract an asynchronous implementation of φ.

Partial results from this work were reported in [PK09], where the approximations $\mathcal{X}_{\downarrow}$ and $\mathcal{X}_{\psi}$ were presented for the case of single input and single output specifications. Beyond extending those results to handle multiple variables, we also prove here some methods that were presented there as conjectures, and correct several mistakes.

3.2 Preliminaries

3.2.1 Temporal Logic

We describe the syntax and semantics of an extension of Quantified Propositional Temporal Logic (QPTL) [SVW87]. QPTL admits quantification over variables. We extend it by *stuttering quantification*, which is explained below. By abuse of notation we use QPTL to refer to this extended logic.

Let X be a set of variables. Without loss of generality we assume that all variables range over the same finite domain D. Let D^X denote the set of functions

from X to D. Given, $d_1, d_2 \in D^X$ and $X' \subseteq X$ we write $d_1 =_{X'} d_2$ if for every $x \in X'$ we have $d_1(x) = d_2(x)$. The syntax of QPTL is defined according to the following grammar.

$$\tau \ ::= \ x = d, \text{ where } x \in X \text{ and } d \in D$$
$$\varphi \ ::= \ \tau \parallel \neg\varphi \parallel \varphi \vee \varphi \parallel \bigcirc \varphi \parallel \ominus \varphi \parallel \varphi \mathcal{U} \varphi \parallel \varphi \mathcal{S} \varphi \parallel (\exists x).\varphi \parallel (\exists^{\approx} x).\varphi$$

where τ are *atomic formulae* and φ are *QPTL formulae* (formulae, for short).

We use the following standard abbreviations (here, $d \in D$, $x, y \in X$, and ψ, ψ_1, ψ_2 are formulae): $x \neq d$ for $\neg(x = d)$, T for $x = d \vee x \neq d$, F for $\neg$T, $\psi_1 \wedge \psi_2$ for $\neg(\neg\psi_1 \vee \neg\psi_2)$, $\psi_1 \rightarrow \psi_2$ for $\neg\psi_1 \vee \psi_2$, $\psi_1 \leftrightarrow \psi_2$ for $(\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1)$, $(\forall x)\psi$ for $\neg(\exists x)(\neg\psi)$, $(\forall^{\approx} x)\psi$ for $\neg(\exists^{\approx} x)(\neg\psi)$, $\diamondsuit \psi$ for T$\mathcal{U}\psi$, $\square \psi$ for $\neg \diamondsuit \neg\psi$, $\diamondminus \psi$ for T $\mathcal{S}$ ψ, $\boxminus \psi$ for $\neg \diamondminus \neg\psi$, $\psi_1 \mathcal{W} \psi_2$ for $\psi_1 \mathcal{U} \psi_2 \vee \square \psi_1$, $\psi_1 \mathcal{B} \psi_2$ for $\psi_1 \mathcal{S} \psi_2 \vee \boxminus \psi_1$, $x = y$ for $\bigvee_{d \in D}(x = d \wedge y = d)$, $x \neq y$ for $\neg(x = y)$, $x = \ominus y$ for $\bigvee_{d \in D}(x = d \wedge \ominus y = d)$, $\circleddash\!\!\sim \psi$ for $\neg \ominus \neg\psi$, and $\psi_1 \Rightarrow \psi_2$ for $\square(\psi_1 \rightarrow \psi_2)$. For a set $\hat{X} = \{x_1, \ldots, x_k\} \subseteq X$ of variables, we write $(\exists \hat{X}).\psi$ for $(\exists x_1) \cdots (\exists x_k).\psi$ and similarly for $(\forall \hat{X}).\psi$. By abuse of notation, we sometimes list variables and sets, e.g., $(\exists \hat{X}, y).\psi$ instead of $(\exists \hat{X} \cup \{y\}).\psi$. Also, in case a Boolean variable, r, we write r for $r = 1$ and $\overline{r}$ for $r = 0$.

Temporal past formulae are QPTL formulae that contain the operators $\mathcal{S}$ or $\ominus$ (or any of the other operators that are expressed with them).

The logic LTL is obtained by disallowing the usage of the $\exists$ and $\exists^{\approx}$ operators. When we want to stress that a formula φ is written over the variables in a set X, we write $\varphi(X)$. When the variables are partitioned to *input variables* (inputs, for short) X and *output variables* (outputs) Y, we write $\varphi(X; Y)$. We call such formulae *specifications*. By abuse of notation, we sometimes list the variables in

X and Y, e.g., $\varphi(x_1, x_2; y)$.

The semantics of QPTL is given with respect to computations and locations in them. A *computation* σ is an infinite sequence $a_0, a_1, \ldots$, where for every $i \geq 0$ we have $a_i \in D^X$. That is, a computation is an infinite sequence of value assignments to the variables in X. A computation $\sigma' = a'_0, a'_1, \ldots$ is an *x-variant* of computation $\sigma = a_0, a_1, \ldots$ if for every $i \geq 0$ and every $y \neq x$ we have $a_i[y] = a'_i[y]$. The computation *squeeze*(σ) is obtained from σ as follows. If for all $i \geq 0$ we have $a_i = a_0$, then *squeeze*$(\sigma) = \sigma$. Otherwise, if $a_0 \neq a_1$ then *squeeze*$(\sigma) = a_0,$ *squeeze*$(a_1, a_2, \ldots)$. Finally, if $a_0 = a_1$ then *squeeze*$(\sigma) =$ *squeeze*$(a_1, a_2, \ldots)$. That is, by removing repeating assignments, *squeeze* returns a computation in which every two adjacent assignments are different unless the computation ends in an infinite suffix of one assignment. A computation σ' is a *stuttering variant* of σ if $\sigma' = a_0^{i_0}, a_1^{i_1}, \ldots$, where *squeeze*$(\sigma) = a_0, a_1, \ldots$ and for every $j \geq 0$ we have $i_j \geq 1$ (here, $a_j^{i_j}$ stands for i_j consecutive repetitions of a_j). For an assignment $a \in X^D$ and a variable $x \in X$ we write $a[x]$ for the value assigned to x by a. If $X = \{x_1, \ldots, x_n\}$, we freely use the notation $(a_{i_1}[x_1], \ldots, a_{i_n}[x_n])$ for the assignment a such that $a[x_j] = a_{i_j}[x_j]$.

Satisfaction of a QPTL formula φ over computation σ in location $i \geq 0$, denoted $\sigma, i \models \varphi$, is defined as follows:

1. For an atomic formula $x = d$, we have $\sigma, i \models x = d$ **iff** $a_i[x] = d$.

2. $\sigma, i \models \neg\varphi$ **iff** $\sigma, i \not\models \varphi$.

3. $\sigma, i \models \varphi \vee \psi$ **iff** $\langle \sigma, i \models \varphi$ or $\sigma, i \models \psi$.

4. $\sigma, i \models \bigcirc\, \varphi$ **iff** $\sigma, i+1 \models \varphi$.

5. $\sigma, i \models \ominus\, \varphi$ **iff** $i > 0$ and $\sigma, i-1 \models \varphi$.

6. $\sigma, i \models \varphi \mathcal{U} \psi$ **iff** for some $j \geq i$, $\sigma, j \models \psi$, and for all k, $i \leq k < j$, $\sigma, k \models \varphi$.

7. $\sigma, i \models \varphi \mathcal{S} \psi$ **iff** for some $j \leq i$, $\sigma, j \models \psi$, and for all k, $i \geq k > j$, $\sigma, k \models \varphi$.

8. For a variable x, we have $\sigma, i \models (\exists x).\varphi$ **iff** $\sigma', i \models \varphi$ for some σ' that is an x-variant of σ.

9. For a variable x we have $\sigma, i \models (\exists^{\approx} x).\varphi$ **iff** $\sigma'', i \models \varphi$ for some σ'' that is a x-variant of some stuttering variant σ' of σ.

We say that the computation σ satisfies the formula φ, **iff** $\sigma, 0 \models \varphi$.

3.2.2 Realizability of Temporal Specifications

We distinguish between synchronous and asynchronous programs. The main difference is in initialization. A synchronous program can initialize its variables based on the values of inputs (that it reads synchronously). An asynchronous program initializes its variables without access to the values of inputs. Let X and Y be the sets of *input* and *output* variables ranging over domain D.

In order to stress the different roles of the system and the environment we specialize the notion of a computation to that of an *interaction*. In an interaction we treat each assignment to $X \cup Y$ as different assignments to X and Y. Thus, instead of using $c \in D^{X \cup Y}$, we use a pair (a, b), where $a \in D^X$ and $b \in D^Y$. Formally, an interaction is $\sigma = (a_0, b_0), (a_1, b_1), \ldots \in (D^X \times D^Y)^\omega$.

Synchronous Programs A *synchronous program* P_s from X to Y is a function $P_s : (D^X)^+ \mapsto D^Y$. In every step of the computation (including the initial one) the program reads its inputs and updates the values of all outputs (based on the entire history). An interaction σ is called *synchronous interaction* of P if, at each

step of the interaction $i = 0, 1, \ldots$, the program outputs (assigns to Y) the value $P_s(a_0, a_1, \ldots, a_i)$, i.e., $b_i = P_s(a_0, \ldots, a_i)$. In such interactions, therefore, both the environment that updates input values and the system that updates output values 'act' at each step (where the system, through the program, responds in each step to an environment action).

A synchronous program is *finite state* if it can be *induced* by a *Labeled Transition System (LTS).* An LTS is $T = \langle S, I, R, X, Y, L\rangle$, where S is a finite set of *states*, $I \subseteq S$ is a set of *initial states*, $R \subseteq S \times S$ is a *transition relation*, X and Y are disjoint sets of *input* and *output* variables, respectively, and $L : S \mapsto D^{X\cup Y}$ is a *labeling* function. For a state $s \in S$ and for $Z \subseteq X \cup Y$, we define $L(s)|_Z$ to be the restriction of $L(s)$ to the variables of Z. The LTS has to be *receptive*, i.e., be able to accept all inputs. Formally, for every $a \in D^X$ there is some $s_0 \in I$ such that $L(s_0)|_X = a$. For every $a \in D^X$ and $s \in S$ there is some $s_a \in S$ such that $R(s, s_a)$ and $L(s_a)|_X = a$. The LTS T is *deterministic* if for every $a \in D^X$ there is a unique $s_0 \in I$ such that $L(s_0)|_X = a$ and for every $a \in D^X$ and every $s \in S$ there is a unique $s_a \in S$ such that $R(s, s_a)$ and $L(s_a)|_X = a$. Otherwise, it is *nondeterministic*. A deterministic LTS T induces the synchronous program $P_T : (D^X)^+ \mapsto D^Y$ as follows. For every $a \in D^X$ let $T(a)$ be the unique state $s_0 \in I$ such that $L(s_0)|_X = a$. For every $n > 1$ and $a_1 \ldots a_n \in (D^X)^+$ let $T(a_1, \ldots, a_n)$ be the unique $s \in S$ such that $R(T(a_1, \ldots, a_{n-1}), s)$ and $L(s)|_X = a_n$. For every $a_1 \ldots a_n \in (D^X)^+$ let $P_T(a_1, \ldots, a_n)$ be the unique $b \in D^Y$ such that $b = L(T(a_1, \ldots, a_n))|_Y$. We note that nondeterministic LTS do not induce programs. As nondeterministic LTS can always be pruned to deterministic LTS, we find it acceptable to produce nondeterministic LTS as a representation of a set of possible programs. We say that the size of T is $|S|$.

Asynchronous Programs An *asynchronous program* P_a from X to Y is a function $P_a : (D^X)^* \mapsto D^Y$. That is, the program sets a value to its output even before seeing the value of inputs. As before, the program receives a new set of inputs and updates its outputs. However, the definition of an interaction takes into account that this may not happen instantaneously.

A *schedule* is a pair (R, W) of sequences $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$ of *reading points* and *writing points* such that $r_1^1 > 0$ and for every $i > 0$ we have $r_i^1 < r_i^2 < \cdots < r_i^n < w_i^1$ and $w_i^1 < w_i^2 < \cdots < w_i^m < r_{i+1}^1$. It identifies the points where each of the input variables is read and the points where each of the output variables is written. The second requirement establishes that reading and writing points occur cyclically. When the distinction is not important, we call reading points and writing points *$I \backslash O$-points*.

An interaction σ is called *asynchronous interaction* of P_a for schedule (R, W) if $b_0 = P_a(\epsilon)$, and for every $i > 0$, every $j \in \{1, \ldots, m\}$, and every k, $w_i^j \leq k < w_{i+1}^j$, we have

$$b_k[j] = P_a((a_{r_1^1}[1], \ldots, a_{r_1^n}[n]), (a_{r_2^1}[1], \ldots, a_{r_2^n}[n]), \ldots, (a_{r_i^1}[1], \ldots, a_{r_i^n}[n]))[j].$$

Also, for every $j \in \{1, \ldots, m\}$ and every $0 < k < w_1^j$, we have that $b_k[j] = b_0[j]$.

In asynchronous interactions, the environment may update the input values at each step. However, the system is only aware of the values of inputs at reading points and responds by outputting the appropriate variables at writing points. In particular, the system is not even aware of the amount of time that passes between the two adjacent time points (read-read, read-write, or write-read). That is, the output values depend only on the values of inputs in earlier reading points.

An asynchronous program is *finite state* if it can *asynchronously induced* by an *Initialized LTS (ILTS)*. An ILTS is $T = \langle T_s, i \rangle$, where $T_s = \langle S, I, R, X, Y, L \rangle$ is

an LTS, and $i \in D^Y$ is an *initial assignment*. We sometimes abuse notations and write $T = \langle S, I, R, X, Y, L, i \rangle$. Determinism is defined just as for LTS. Similarly, given $a_1, \ldots, a_n \in (D^X)^+$ we define $T(a_1, \ldots, a_n)$ as before. A deterministic ILTS T asynchronously induces the program $P_T : (D^X)^* \mapsto D^Y$ as follows. Let $P_T(\epsilon) = i$ and for every $a_1 \ldots a_n \in (D^X)^+$ we have $P_T(a_1, \ldots, a_n)$ as before. By choosing i to be a unique initial assignment, we force ILTS to induce only asynchronous programs that deterministically assign a single initial value to outputs. This definition could be expanded to allow a nondeterministic choice of initial output values (so long as they need not depend on inputs which are unavailable), and all the results presented in this work would still hold. For simplicity and clarity we choose the definition above, and we leave it to readers to expand (since programs are induced by deterministic ILTS, this choice is even more reasonable).

Definition 3.1 (realizability)**.** *A* LTL *specification $\varphi(X;Y)$ is* **synchronously realizable** *if there exists a synchronous program P_s such that all synchronous interactions of P_s satisfy $\varphi(X;Y)$. Such a program P_s is said to* synchronously realize *$\varphi(X;Y)$.* Synchronous realizability *is often simply shortened to* realizability*.*

The specification $\varphi(X;Y)$ is **asynchronously realizable** *if there exists an asynchronous program P_a such that all asynchronous interactions of P_a (for all schedules) satisfy $\varphi(X;Y)$. Such a program P_a is said to* asynchronously realize *$\varphi(X;Y)$.*

Synthesis is the process of automatically constructing a program P that (synchronously/asynchronously) realizes a specification $\varphi(X;Y)$. We freely write that an LTS realizes a specification in case that the induced program satisfies it.

The following theorem is proven in [PR89a].

Theorem 3.1 ([PR89a])**.** *Both of the following hold:*

1. *The problem of deciding whether a specification $\varphi(X;Y)$ is synchronously realizable is 2EXPTIME-complete.*

2. *Given a synchronously realizable specification $\varphi(X;Y)$ there exists a doubly-exponential algorithm to construct a LTS that synchronously realizes $\varphi(X;Y)$.*

3.2.3 Structure and Notations of Specifications

In this subsection we define some normal form of specifications that we treat. Let X and Y be disjoint sets of input and output variables, respectively. We consider specifications that describe an interplay between a *system* s and an *environment* e. The specification has two parts, for $\alpha \in \{e, s\}$, $\varphi_\alpha(X, Y)$, which is the formula that defines the allowed actions of α is a conjunction of:

1. I_α (*initial condition*) – a Boolean formula (equally, an assertion) over $X \cup Y$, describing the initial state of α. The formula I_s may refer to all variables and I_e may refer only to the variables X;

2. $\Box\, S_\alpha$ (*safety component*) – a formula describing the transition relation of α, where S_α describes the update of the locally controlled state variables (identified by being *primed*, e.g., x' for $x \in X$) as related to the current state (unprimed, e.g., x), with the exception that s can observe X's next values;

3. L_α (*liveness component*) – each L_α is a conjunction of $\Box \Diamond p$ formulae where p is a Boolean formula.

In the case that a specification includes temporal past formulae instead of the Boolean formulae in any of the three conjuncts mentioned above, we assume that a pre-processing of the specification was done to translate it into another one that has the same structure but without the use of past formulae. This can be

always achieved through the introduction of fresh Boolean variables that implement temporal testers for past formulae [PZ08]. Therefore, without loss of generality, we discuss in this work only such past-formulae-free specifications.

We abuse notations and write φ_α also as a triplet $\langle I_\alpha, S_\alpha, L_\alpha \rangle$.

Consider a pair of formulae $\varphi_\alpha(X, Y)$, for $\alpha \in \{e, s\}$, where $\varphi_\alpha = I_\alpha \wedge \Box S_\alpha \wedge L_\alpha$. We define a structure of specification formulae:

$$Imp(\varphi_e, \varphi_s) : (I_e \wedge \Box S_e \wedge L_e) \rightarrow (I_s \wedge \Box S_s \wedge L_s)$$

For such specifications, the *winning condition* is defined to be the formula $L_e \rightarrow L_s$.

LTL formulae are called *Generalized Reactivity (1) ($GR(1)$)* formulae if they have the form

$$(\Box \Diamond p_1 \wedge \ldots \wedge \Box \Diamond p_m) \;\rightarrow\; (\Box \Diamond q_1 \wedge \ldots \wedge \Box \Diamond q_n),$$

where all p_i and q_j are Boolean formulae. *Generalized Reactivity (k) ($GR(k)$)* formulae are conjunctions of k $GR(1)$ formulae.

In this work, we often concentrate on specifications φ that have the form $Imp(\varphi_e, \varphi_s)$ and that their winning condition $L_e \rightarrow L_s$ is in $GR(1)$. This is due to the fact that we propose using the algorithms described in [PP06, PPS06] that test such specifications for synchronous realizability and synthesis.

3.2.4 The Rosner Reduction

In [PR89b], Pnueli and Rosner show how to use synchronous realizability to solve asynchronous realizability. They define, what we call, *the Rosner reduction*. It translates a specification $\varphi(X; Y)$, where $X = \{x\}$ and $Y = \{y\}$ are singletons,

into a specification $\mathcal{X}(x, r; y)$ that has an additional Boolean input variable r. The new variable r is called the *Boolean scheduling variable*. Intuitively, the Boolean scheduling variable defines all possible schedules for one input-one output systems. When it changes from zero to one it signals a reading point and when it changes from one to zero it signals a writing point. Formally, given a specification $\varphi(x; y)$, the reduction defines the *kernel formula* $\mathcal{X}(x, r; y)$:

$$\underbrace{\overline{r} \wedge \Box \Diamond r \wedge \Box \Diamond \overline{r}}_{\alpha(r)} \rightarrow \underbrace{\begin{pmatrix} \varphi(x; y) & \wedge \\ (r \vee \circleddash \overline{r}) \Rightarrow (y = \circleddash y) & \wedge \\ (\forall^{\approx} \widetilde{x}).[(r \wedge \circleddash \overline{r}) \Rightarrow (x = \widetilde{x})] \rightarrow \varphi(\widetilde{x}; y) & \end{pmatrix}}_{\beta(x,r;y)}$$

The new specification is an implication between $\alpha(r)$ and $\beta(x, r; y)$. The formula $\alpha(r)$ governs the behavior of r. It dictates that the first $I \backslash O$-point is a reading point (r changing from zero to one) and that there are infinitely many reading points and infinitely many writing points. Then, $\beta(x, r; y)$ includes three parts. First, the original formula $\varphi(x; y)$ must hold. Second, outputs behave according to the signals given by the scheduling variable. That is, in all points that are not writing points the value of y does not change. Third, we use the stuttering quantification to say that if we replace all the inputs except in reading points, then the same output still causes the original formula to be satisfied[7].

The following theorem is proven in [PR89b].

Theorem 3.2 ([PR89b]). *The specification $\varphi(x; y)$ is asynchronously realizable* **iff** *the kernel formula $\mathcal{X}(x, r; y)$, which is derived from $\varphi(x; y)$ using the Ronser*

[7] The first conjunct of $\beta(x, r; y)$, $\varphi(x; y)$, is redundant. It is a consequence of the third conjunct which guarantees that $\varphi(\tilde{x}; y)$ is satisfied for a set of sequences of assignments to $\tilde{x}$ which includes the single sequence of assignments to x. We leave this conjunct here, as well as in similar reductions later in this chapter, for clarity.

reduction, is synchronously realizable.

Furthermore, given a program P_s that synchronously realizes $\mathcal{X}(x, r; y)$ it can be converted to a program P_a that asynchronously realizes $\varphi(x; y)$ in time linear in the number of transitions of the LTS that induces P_s.

Pnueli and Rosner also show that an extension of the realizability technique for LTL can be used for formulae that contain $\forall^{\approx}\tilde{x}$ quantification of the form used in $\mathcal{X}(x, r; y)$.

3.3 Expanding the Rosner Reduction to Multiple Variables

In this section we describe an expansion of the Rosner reduction to handle specifications with multiple inputs and output variables. This new reduction achieves the same desired outcome of reducing the problem of asynchronous synthesis to that of synchronous synthesis. For this section, without loss of generality, fix an LTL specification $\varphi(X; Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$.

We propose a family of translations (one translation for each pair (n, m)) called the *generalized Rosner reduction*. These translations translate $\varphi(X; Y)$ into a QPTL specification $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. The specification uses an additional input variable r, called the *scheduling variable*, that ranges over $\{1, \ldots, (n + m)\}$.

As in the Rosner reduction, the scheduling variable defines all reading and writing points. Variable x_i may be read by the system whenever r changes its value to i. Variable y_i may be modified whenever r changes to equal $n + i$. Initially, $r = n + m$ and it can only be cyclically incremented by 1 (hence, the variable x_1 is the first variable that is read, in the first $I\backslash O$-point). Let $i \oplus_m 1$ denote $(i \; mod \; m) + 1$.

We also denote $[r = (n + i)] \wedge \ominus[r \neq (n + i)]$ by $write_n(i)$ to indicate a state

that is a writing point for y_i, $(r = i) \wedge \ominus(r \neq i)$ by $read(i)$ to indicate a state that is a reading point for x_i, $\bigwedge_{d \in D(z)}[(z = d) \leftrightarrow \ominus(z = d)]$ by $unchanged(z)$ to indicate a state where z did not change its value ($D(z)$ denotes the domain of variable z), and $\neg \ominus \top$ by *first* to indicate a state that is the first one in the computation. Note that in $write_n(i)$ and in $read(i)$ we do not demand explicitly that the scheduling variable held in the previous state the appropriate value in its 'cycle'. Rather, we simply require that its previous value is different from the current one. The appropriate clause in our reduction, that describes the update of the scheduling variable, guarantees that there would be only one possible assignment to this variable in the previous state, given its current value and the fact that they differ.

The kernel formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is defined by

$$\mathcal{X}^{n,m}(X \cup \{r\}; Y) = \alpha^{n,m}(r) \to \beta^{n,m}(X \cup \{r\}; Y)$$

where

$$\alpha^{n,m}(r) = \begin{pmatrix} r = (n+m) & \wedge \\ \bigwedge_{i=1}^{n+m} \Big[(r = i) \Rightarrow \big[(r = i) \mathcal{U} [r = (i \oplus_{n+m} 1)] \big] \Big] & \end{pmatrix}$$

and $\beta^{n,m}(X \cup \{r\}; Y)$ is given by

$$\begin{pmatrix} \varphi(X; Y) & \wedge \\ \bigwedge_{i=1}^{m} \Big[[\neg write_n(i) \wedge \neg first] \Rightarrow unchanged(y_i) \Big] & \wedge \\ (\forall^{\approx} \tilde{X}). \Big[\bigwedge_{i=1}^{n} [read(i) \Rightarrow (x_i = \tilde{x}_i)] \Big] \to \varphi(\tilde{X}; Y) & \end{pmatrix}.$$

The clause *first* is added to avoid the following problem: due to the specific logic

of the clause *unchanged*(z), for any variable z, it always evaluates to F in the first state, often causing the entire specification to become F. This is an unwanted effect, and any clauses that describe a condition for some variable to not change its value (e.g., *unchanged*(z)), have no importance in the first state anyway. The Rosner reduction does not need a similar clause since its logic does not cause this abnormality.

There is a 1-1 and onto correspondence between a sequence of assignments to r and a schedule (R, W). That is, given a sequence of values $r_0, r_1, \ldots$ that satisfies $\alpha^{n,m}$ it defines the *implied schedule* (R, W) by the sequences
$R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$, where

- r_0^j is the minimal k such that $r_k = j$ and $r_{k-1} \neq j$ and r_{i+1}^j is the minimal k larger than r_i^j such that $r_k = j$ and $r_{k-1} \neq j$.
- w_0^j is the minimal k such that $w_k = n + j$ and $w_{k-1} \neq n + j$ and w_{i+1}^j is the minimal k larger than w_i^j such that $w_k = n + j$ and $w_{k-1} \neq n + j$.

Clearly, a similar correspondence can be constructed in the other direction.

Intuitively, as r is an input variable, the program has to handle all possible assignments to it. Then, the correspondence between a sequence of assignments to r (that satisfies $\alpha^{n,m}$) and a schedule, explains how satisfying $\mathcal{X}^{n,m}$ would be sufficient to handle every possible schedule.

The following theorem expands Theorem 3.2 to the case of multiple-variable specifications:

Theorem 3.3. *The specification $\varphi(X; Y)$ ($|X| = n$, and $|Y| = m$) is asynchronously realizable* **iff** *the kernel formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, which is derived from $\varphi(X; Y)$ using the generalized Ronser reduction, is synchronously realizable.*

Furthermore, given a program P_s that synchronously realizes $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ it can be converted to a program P_a that asynchronously realizes $\varphi(X;Y)$ in time linear in the number of transitions of the LTS that induces P_s, and vice versa.

Proof: For clarity, we define $D_r = \{1, \ldots, (n+m)\}$ (the domain of the scheduling variable r). We also use the notation $r_{init} = n+m$ for the 'correct' initial value of r. We shall prove both directions constructively, by reducing each type of program to the other:

$\Leftarrow$ Let $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ be synchronously realized by the synchronous program $P_s : (D^X \times D_r)^+ \mapsto D^Y$. We define the asynchronous program $P_a : (D^X)^* \mapsto D^Y$ as shown below.

$P_a(\epsilon) = P_s((a_{rand}, r_{init}))$, where a_{rand} is some arbitrary assignment to the inputs X. We define the function $dup_{n,m} : (D^X)^+ \mapsto (D^X)^+$ inductively: For all $a \in D^X$, $dup_{n,m}(a) = a^1, \ldots, a^n, \underbrace{a^n, \ldots, a^n}_{m\ times}$ where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ $a^i[j] = a[j]$ and for all j such that $i < j \leq n$ $a^i[j] = a_{rand}[j]$. For all $k > 1$ and $a_1, \ldots, a_k \in (D^X)^k$, $dup_{n,m}(a_1, \ldots, a_k) = dup_{n,m}(a_1, \ldots, a_{k-1}), a'^1, \ldots, a'^n, \underbrace{a'^n, \ldots, a'^n}_{m\ times}$ where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ $a'^i[j] = a_k[j]$ and for all j such that $i < j \leq n$ $a'^i[j] = a_{k-1}[j]$. For all $k > 0$ let $r^k = r_1, \ldots, r_k$, where $r_1 = 1$ ($r_1 = r_{init} \oplus_{n+m} 1$), and where for all $k \geq i > 1$, $r_i = r_{i-1} \oplus_{n+m} 1$. For all $k > 0$, if $a_1, \ldots, a_k \in (D^X)^k$ we define $dupr_{n,m}(a_1, \ldots, a_k) \in (D^X \times D_r)^{(n+m)\cdot k}$ where the projection of $dupr_{n,m}(a_1, \ldots, a_k)$ on the inputs X is $dup_{n,m}(a_1, \ldots, a_k)$, and the projection of $dupr_{n,m}(a_1, \ldots, a_k)$ on the scheduling variable r is $r^{(n+m)\cdot k}$. Finally, for all $k > 0$ such that $a_1, \ldots, a_k \in (D^X)^k$ we define $P_a(a_1, \ldots, a_k) = P_s((a_{rand}, r_{init}), dupr_{n,m}(a_1, \ldots, a_k))$.

We now show that all asynchronous interactions of P_a, for all schedules, satisfy $\varphi(X;Y)$, implying that $\varphi(X;Y)$ is asynchronously realized by P_a. Let (R, W) be a schedule, and let $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be an asynchronous interaction of P_a for this schedule. Denote $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$.

We abuse notation and define the function $dup_{n,m} : (D^X \times D^Y)^+ \mapsto (D^X \times D^Y)^+$ inductively: For all $(a, b) \in D^X \times D^Y$,

$$dup_{n,m}((a,b)) = (a^1, b_0), \ldots, (a^n, b_0), (a^n, b^1), \ldots, (a^n, b^m)$$

where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ $a^i[j] = a[j]$ and for all j such that $i < j \leq n$ $a^i[j] = a_{rand}[j]$. Also, for all $0 < i \leq m$, for all j such that $0 < j \leq i$ $b^i[j] = b[j]$ and for all j such that $i < j \leq m$ $b^i[j] = b_0[j]$. For all $k > 1$ and $(c_1, d_1), \ldots, (c_k, d_k) \in (D^X \times D^Y)^k$,

$$\begin{aligned} &dup_{n,m}((c_1, d_1), \ldots, (c_k, d_k)) = \\ &\qquad dup_{n,m}\Big((c_1, d_1), \ldots, (c_{k-1}, d_{k-1})\Big), \\ &\qquad (a^1, d_{k-1}), \ldots, (a^n, d_{k-1}), (a^n, b^1), \ldots, (a^n, b^m) \end{aligned}$$

where for all $0 < i \leq n$, for all j such that $0 < j \leq i$ $a^i[j] = c_k[j]$ and for all j such that $i < j \leq n$ $a^i[j] = c_{k-1}[j]$. Also, for all $0 < i \leq m$, for all j such that $0 < j \leq i$ $b^i[j] = d_k[j]$ and for all j such that $i < j \leq m$ $b^i[j] = d_{k-1}[j]$. We also abuse notation by defining that for all $k > 0$, if $(c_1, d_1), \ldots, (c_k, d_k) \in (D^X \times D^Y)^k$ $dupr_{n,m}((c_1, d_1), \ldots, (c_k, d_k)) \in (D^X \times D_r \times D^Y)^{(n+m)\cdot k}$ where the projection of $dupr_{n,m}((c_1, d_1), \ldots, (c_k, d_k))$ on the inputs and outputs $X \cup Y$ is $dup_{n,m}((c_1, d_1), \ldots, (c_k, d_k))$, and the projection

of $dupr_{n,m}((c_1, d_1), \ldots, (c_k, d_k))$ on the scheduling variable r is $r^{(n+m)\cdot k}$. We also apply *dup* and *dupr* to infinite computations. In that case, the result is the limit of the application of the function on all prefixes of the infinite computation.

Consider the computation

$$\begin{aligned}\sigma' = \; & \Big((a_{r_1^1}[1], \ldots, a_{r_1^n}[n]), (b_{w_1^1}[1], \ldots, b_{w_1^m}[m])\Big), \\ & \Big((a_{r_2^1}[1], \ldots, a_{r_2^n}[n]), (b_{w_2^1}[1], \ldots, b_{w_2^m}[m])\Big), \ldots\end{aligned}$$

obtained from σ by restricting attention to the values of variables to the appropriate reading and writing points. By construction, we know that the computation $\sigma'' = (a_{rand}, r_{init}, b_0), dupr_{n,m}(\sigma')$ is a synchronous interaction of P_s. Therefore, $\sigma'', 0 \models \mathcal{X}^{n,m}$. Since $\sigma'', 0 \models \alpha^{n,m}$ (due to the way $dupr_{n,m}$ modifies the scheduling variable), we also get that

$$\sigma'', 0 \models (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} [read(i) \Rightarrow (x_i = \tilde{x}_i)] \rightarrow \varphi(\tilde{X}; Y). \tag{3.1}$$

We now define the computation σ''', which is obtained from σ'' by 'stretching' it so that all reading points in σ''' match exactly with the indices of R, and all writing points in it match exactly with the indices of W. By 'matching exactly' we mean that there are no $I\backslash O$ points in σ''' beyond those indicated by the schedule (R, W). When we stretch σ'', the newly added states are copies of their predecessor (all newly added states in the middle of s, t are duplicates of s). As a result, the first state in σ''' is exactly the first state in σ'' – $(a_{rand}, r_{init}, b_0)$ – and there are exactly r_1^1 copies of it at the prefix of σ'''. The second state of σ'' is duplicated to $r_1^2 - r_1^1$ copies in σ''', the n-th

state is duplicated to $r_1^n - r_1^{n-1}$ copies, the $n+1$-th state is duplicated to $w_1^1 - r_1^n$ copies, the $n+m$-th state is duplicated to $w_1^m - w_1^{m-1}$ copies, the $n+m+1$-th state is duplicated to $r_2^1 - w_1^m$ copies, and so on.

From Formula 3.1, every stuttering variant of σ'' that assigns to $\tilde{X}$ values that agree with X in all reading points, satisfies $\varphi(\tilde{X};Y)$. This is exactly the case of σ. Indeed, σ''' is a stuttering variant of σ'' and agrees with σ on assignments to the outputs Y and to r. It follows that if we consider the assignment of σ to X as an assignment to $\tilde{X}$ added to σ''', we get the required result that $\sigma, 0 \models \varphi(X,Y)$ which concludes the proof of this direction.

$\Rightarrow$ Let $\varphi(X;Y)$ be asynchronously realized by the asynchronous program $P_a : (D^X)^* \mapsto D^Y$. We define the synchronous program $P_s : (D^X \times D_r)^+ \mapsto D^Y$ as shown below.

For all $k > 0$ and $a_1, \ldots, a_k \in (D^X \times D_r)^k$, if, denoting r_i as the value of r in a_i, $r_1 \neq r_{init}$ or that there exists some index $1 < j \leq n$ such that $(r_j \neq r_{j-1}) \wedge (r_j \neq r_{j-1} \oplus_{n+m} 1)$ holds, then $P_s(a_1, \ldots, a_k, \ldots) = b_{rand}$ (for all prefixes of computations with the sub-prefix $a_1, \ldots, a_k$), where b_{rand} is some arbitrary assignment to the outputs Y. From this point onwards, we handle only elements of $(D^X \times D_r)^k$ for which the above condition does not hold and which are, therefore, 'compliant' with the initial condition and transition relation implied by $\alpha^{n,m}(r)$. It is worthwhile to note that monitoring this condition could be done 'on-line' while P_s gets more and more inputs, without any increase in complexity.

For all $k > 0$ and given $a_1, \ldots, a_k \in (D^X \times D_r)^k$, we define the *implied prefixed schedule* for $a_1, \ldots, a_k$, (R^k, W^k), to be a prefix of some sched-

ule implied by some extension of $a_1, \ldots, a_k$ to a computation that satisfies $\alpha^{n,m}$ (i.e., the sequences R^k and W^k could be extended infinitely to such a schedule). For (R^k, W^k) to be the (unique) prefixed schedule implied by $a_1, \ldots, a_k$, we require that $|R^k| + |W^k|$ equals the number of times r changes its value in $a_1, \ldots, a_k$. Therefore, (R^k, W^k) represents exactly all the $I \backslash O$ points of $a_1, \ldots, a_k$. For all $k > 1$ and $a_1, \ldots, a_k \in (D^X \times D_r)^k$, if (R^k, W^k) is the prefixed schedule implied by $a_1, \ldots, a_k$, let $R^k = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots, r_t^1, \ldots, r_t^s$ and
$W^k = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots, w_p^1, \ldots, w_p^q$. Let c_i (for all i) be the projection of a_i on the inputs X. We define $b_{prev(s,t)} \in D^Y$ to be the value assigned by P_a to the outputs at the end of the previous-to-most-recent 'full $I \backslash O$ cycle':

$$b_{prev(s,t)} = P_a((c_{r_1^1}[1], \ldots, c_{r_1^n}[n]), (c_{r_2^1}[1], \ldots, c_{r_2^n}[n]), \ldots, (c_{r_g^1}[1], \ldots, c_{r_g^n}[n]))$$

where if $s = n$ then $g = t - 1$ and otherwise $g = t - 2$. We also define $b_{last(s,t)} \in D^Y$ to be the value assigned by P_a to the outputs at the end of the most recent 'full $I \backslash O$ cycle':

$$b_{last(s,t)} = P_a((c_{r_1^1}[1], \ldots, c_{r_1^n}[n]), (c_{r_2^1}[1], \ldots, c_{r_2^n}[n]), \ldots, (c_{r_f^1}[1], \ldots, c_{r_f^n}[n]))$$

where if $s = n$ then $f = t$ and otherwise $f = t - 1$. Note that in some cases (when t is 'too small'), $b_{prev(s,t)} = P_a(\epsilon)$ or $b_{last(s,t)} = P_a(\epsilon)$. The output of P_s, $P_s(a_1, \ldots, a_k)$, should always be some combination of $b_{last(s,t)}$ and $b_{prev(s,t)}$, based on the output variables of Y that were already updated in the most recent 'writing cycle' as indicated by w_p^q. Hence, $P_s(a_1, \ldots, a_k) = b_{real(s,t,q)}$

where for all $0 < i \leq m$, if $i > q$ then $b_{real(s,t,q)}[i] = b_{prev(s,t)}[i]$, and otherwise $b_{real(s,t,q)}[i] = b_{last(s,t)}[i]$. Note that as a result of this definition of P_s, as long as $a_1, \ldots, a_k$ contains no $I \backslash O$ points, $P_s(a_1, \ldots, a_k) = P_a(\epsilon)$. Particularly, for all $a \in D^X \times D_r$, $P_s(a) = P_a(\epsilon)$.

We now show that all synchronous interactions of P_s satisfy $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, implying that $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realized by P_s. Let $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be a synchronous interaction of P_s.

If $\sigma, 0 \not\models \alpha^{n,m}(r)$, then trivially $\sigma, 0 \models \mathcal{X}^{n,m}$ and we are done. Otherwise, we observe that all computations σ' that are $\tilde{X}$- variants of stuttering variants of σ, in which X and $\tilde{X}$ agree in all reading points, are asynchronous interactions of P_a for the schedule implied by the values of r in σ'. Hence, by correctness of P_a, for every such σ' it holds that $\sigma', 0 \models \varphi(\tilde{X}; Y)$. It follows that $\sigma, 0 \models (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} [read(i) \Rightarrow\!\!\!- (x_i = \tilde{x_i})] \rightarrow \varphi(\tilde{X}; Y)$ and, particularly, also that $\sigma, 0 \models \varphi(X; Y)$. By construction, $\sigma, 0 \models \bigwedge_{i=1}^{m} [\neg write_n(i) \wedge \neg first] \Rightarrow\!\!\!- unchanged(y_i)$ and we finally conclude (given that σ satisfies $\alpha^{n,m}(r)$ as well as all of the three conjuncts on the right-hand-side of $\mathcal{X}^{n,m}$), that $\sigma, 0 \models \mathcal{X}^{n,m}(X \cup \{r\}; Y)$. This concludes the proof.

$\square$

In principle, this theorem provides a complete solution to the problem of asynchronous synthesis (with multiple inputs and outputs). Given a specification $\varphi(X; Y)$, we derive for it the kernel formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, and then apply the synchronous synthesis algorithms as described in [PR89a]. By Theorem 3.3 we can deduce from that on the asynchronous realizability of $\varphi(X; Y)$ and, if possible, construct an asynchronously realizing program for it. However, the approach de-

scribed in [PR89a] requires to construct a deterministic automaton for $\mathcal{X}^{n,m}$ and then to solve parity or Rabin games of high topological complexity. In particular, when combining determinization with the treatment of $\forall^{\approx}$ quantification, even relatively simple specifications may lead to very complex deterministic automata and (as a result) games that are complicated to solve.

The main culprit is the third conjunct

$$\beta_3^{n,m} = (\forall^{\approx}\tilde{X}).\Big[\bigwedge_{i=1}^{n}[read(i) \Rightarrow (x_i = \tilde{x}_i)]\Big] \rightarrow \varphi(\tilde{X};Y)$$

which includes the universal quantification over $\tilde{X}$. The algorithm in [PR89b] for handling synthesis of formulas of the form $(\forall^{\approx}z).\psi(z)$ is as follows. It starts by constructing a nondeterministic Büchi automaton for $\psi(z)$. Then, transitions that factor in stuttering are added to this automaton. Finally, the information regarding z is projected. It follows that even if the automaton for $\psi(z)$ is relatively simple, or, indeed, deterministic, the resulting automaton after this procedure is nondeterministic to a high degree. Then, in order to use this automaton in synthesis it has to be determinized. In practice, determinization has been impossible to implement and leads to system-environment games with winning conditions that fall very high in the reactivity hierarchy presented in Fig. 3.1. As explained in [PP06], if a game has a winning condition that falls in the class $GR(k)$, then the synthesis algorithm costs $O(N^{k+2} \cdot k!)$ time, where N is the *state space* of the specification ($|D|^{|X|+|Y|}$ in our case). The combination of determinization and synthesis algorithms for $GR(k)$ specifications has turned out to be impractical. It follows that the theoretical solution suggested in [PR89b] (and extended here) is of little practical value. In particular, the stuttering quantification over $\tilde{X}$ –

$(\forall^{\approx}\tilde{X})$ – makes this full treatment required even in cases where $\varphi(X;Y)$ is relatively simple, for example if $\varphi(X;Y)$ has a $GR(1)$ winning condition as in [PPS06]. Furthermore, in order to handle the stuttering quantification *all* the formula has to be handled together, making the structural treatment of simple specification in [PPS06] impossible. It follows that unlike synchronous synthesis where $GR(1)$ or $GR(2)$ specifications can be synthesized 'effectively' (i.e., low-order polynomial in the size of the state space) as in [PPS06, PP06], in the case of asynchronous synthesis there are no effective algorithms.

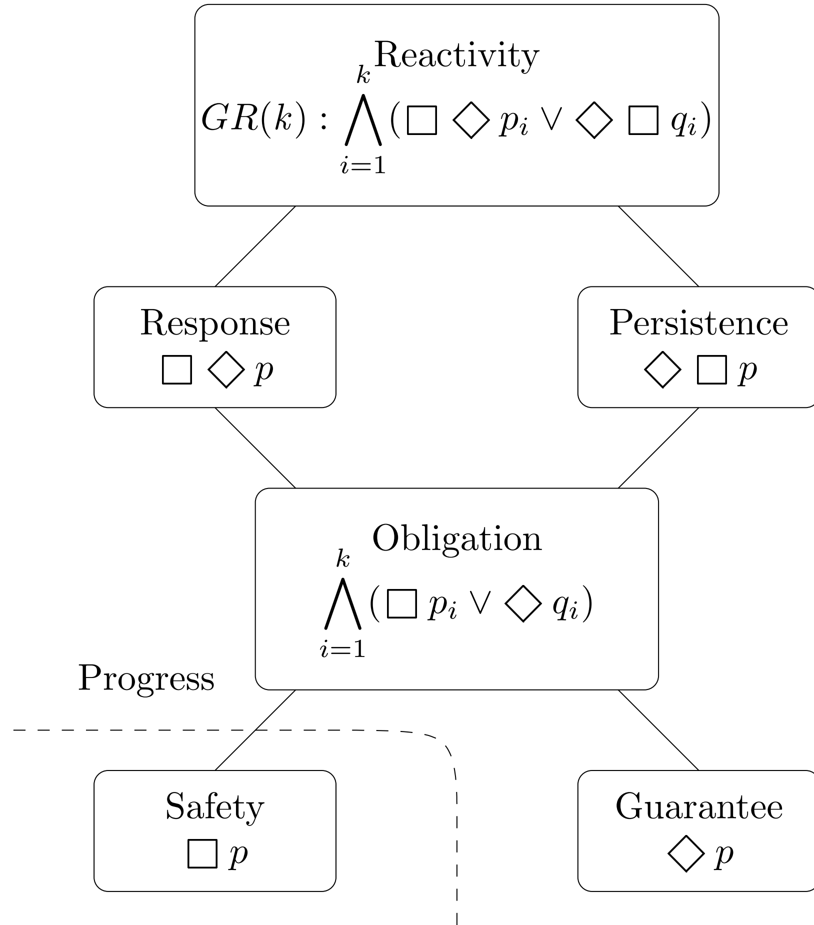


Figure 3.1: The temporal hierarchy of properties

An interesting consequence of this reduction, which is also true for the Rosner reduction, is that any specification $\varphi(X;Y) = Imp(\varphi_e, \varphi_s)$, where $\varphi_s = \langle I_s, S_s, L_s \rangle$, is asynchronously unrealizable if I_s is a function of X. This is since $\mathcal{X}^{n,m}$ does not allow, let alone guarantee, that the first state in a computation is a reading point where $X = \tilde{X}$.

3.4 A More General Asynchronous Interaction Model

The reader may object to the model of asynchronous interaction as over simplified. Here, we justify this model by showing that it is practically equivalent (from a synthesis point of view) to a model that is more akin to software thread implementation. Specifically, we introduce a more elaborate model of asynchronous interaction. In this model the environment chooses the times at which the system can read or write and the system chooses whether to read or write a variable and which variable it wants to access. We start by formally defining this asynchronous interaction. We then show how to solve synthesis for it and how a synthesized solution to the first model induces a solution to the more intricate model. For simplicity, we call our original asynchronous interaction model *round robin* and the new model *by demand*.

For this section, without loss of generality, fix an LTL specification $\varphi(X;Y)$, where $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$.

3.4.1 A General (Multi-Core) Model

Here we describe a more general model of asynchronous interaction. As mentioned, the model allows the environment to choose the times at which the system acts. The system chooses what kind of actions to do; Either read the value of one of the

input variables or write the value of one of the output variables. However, this requires us to define a more elaborate model of a program. Thus, for the purposes of this section, we define a version of a program that is tailored for by-demand asynchronous interaction. In the rest of the chapter, we use the models of synchronous and asynchronous programs, which are very similar. This decision simplifies the transformations between a synchronous program and an asynchronous program.

A *by-demand program* P_b from X to Y is a function $P_b : D^* \mapsto \{1, \ldots, n\} \cup (D \times \{n+1, \ldots, n+m\})$. We assume that for $0 \leq i < m$ and for every $d_1, \ldots, d_{m-1} \in D$, we have $P_b(d_1, \ldots, d_i) = (d, (n + i + 1))$ for some $d \in D$. That is, for a given history of values read\written by the program (and the program should know which variables it read\wrote) the program decides on the next variable to read\write. In case that the decision is to write in the next $I \backslash O$ point, the program also chooses the value to write. Furthermore, the program starts by writing all the output variables according to their order $y_1, y_2, \ldots, y_m$.

By demand asynchronous interaction Recall that an interaction over X and Y is $\sigma = (a_0, b_0), (a_1, b_1), \ldots \in (D^X \times D^Y)$. An *$I \backslash O$-sequence* is $C = c_0, c_1, \ldots$ where $0 = c_0 < c_1 < c_2, \ldots$. It identifies the points in which the program reads or writes. For a sequence $d_1, \ldots, d_k \in D^*$, we denote by $t(P_b(d_1, \ldots, d_k))$ the value j such that either $P_b(d_1, \ldots, d_k) \in \{1, \ldots, n\}$ and $P_b(d_1, \ldots, d_k) = j$ or $P_b(d_1, \ldots, d_k) \in D \times \{n + 1, \ldots, n + m\}$ and $P_b(d_1, \ldots, d_k) = (d, j)$. Given an interaction σ, an $I \backslash O$ sequence C, and an index $i \geq 0$, we define the *view* of P_b,

denoted $v(P_b, \sigma, C, i)$, as follows.

$$v(P_b, \sigma, C, i) = \begin{cases} b_0[1], \ldots, b_0[m] & \text{If } i = 0 \\ v(P_b, \sigma, C, i-1), a_{c_i}[t(P_b(v(P_b, \sigma, C, i-1)))] & \\ \qquad \text{If } i > 0 \text{ and } t(P_b(v(P_b, \sigma, C, i-1))) \leq n & \\ v(P_b, \sigma, C, i-1), b_{c_i}[t(P_b(v(P_b, \sigma, C, i-1)))] & \\ \qquad \text{If } i > 0 \text{ and } t(P_b(v(P_b, \sigma, c, i-1))) > n & \end{cases}$$

Initially, the view of the program includes the values of all outputs at time zero. The view at point c_i extends the view at point c_{i-1} by adding the value of the variable that the program decides to read\write based on its view at point c_{i-1}.

The interaction σ is a *by-demand asynchronous interaction* of P_b for $I\backslash O$ sequence C if for every $1 \leq j \leq m$ we have $P_b(b_0[1], \ldots, b_0[j-1]) = (b_0[j], (n+j))$, and for every $i > 1$ and every $k > 0$ such that $c_i \leq k < c_{i+1}$, we have

- If $t(P_b(v(P_b, \sigma, C, i-1))) \leq n$ then for every $j \in \{1, \ldots, m\}$ we have $b_k[j] = b_{k-1}[j]$.
- If $t(P_b(v(P_b, \sigma, C, i-1))) > n$ then for every $j \neq t(P_b(v(P_b, \sigma, C, i-1)))$ we have $b_k[j] = b_{k-1}[j]$ and for $j = t(P_b(v(P_b, \sigma, C, i-1)))$ we have $P_b(v(P_b, \sigma, c, i-1)) = (b_k[j], j)$.

Also, for every $j \in \{1, \ldots, m\}$ and every $0 < k < c_1$, we have $b_k[j] = b_0[j]$.

That is, the program starts by writing all outputs. Then, it keeps outputs constant except at $I\backslash O$ points where it chooses to update a specific output.

Definition 3.2 (by-demand realizability). *A* LTL *specification* $\varphi(X;Y)$ *is* **by-demand asynchronously realizable** *if there exists a by-demand program* P_a *such that all by-demand asynchronous interactions of* P_a *(for all* $I\backslash O$*-sequences)*

satisfy $\varphi(X;Y)$*. Such a program* P_a *is said to* by-demand asynchronously realize $\varphi(X;Y)$*.*

Theorem 3.4. *Let* $\varphi(X;Y)$ *be a* LTL *specification where* $|X| = n$ *and* $|Y| = m$*.*

$\varphi(X;Y)$ *is asynchronously realizable* **iff** *it is by-demand asynchronously realizable.*

Furthermore, given a program P_a *that asynchronously realizes* $\varphi(X;Y)$*, it can be converted to a program* P_b *that by-demand asynchronously realizes* $\varphi(X;Y)$ *in time linear in the number of transitions of the ILTS that induces* P_a*, and vice versa.*

Proof: We shall prove both directions:

$\Leftarrow$ Let $\varphi(X;Y)$ be by-demand asynchronously realized by the by-demand asynchronous program $P_b : D^* \mapsto \{1,\ldots,n\} \cup (D \times \{n+1,\ldots,n+m\})$. We define the asynchronous program $P_a : (D^X)^* \mapsto D^Y$ as shown below.

For all $k \geq 0$, $a_1,\ldots,a_k \in (D^X)^k$, we define $P_a(a_1,\ldots,a_k)$ inductively, as follows. We also define inductively $v_k \in D^+$, which holds the k-th view of P_b that is used to define P_a. Set $P_a(\epsilon) = b_0$, where for all $0 < i \leq m$ $P_b(b_0[1],\ldots,b_0[i-1]) = (b_0[i],(n+i))$ (this uniquely defines $b_0 \in D^Y$). Also, let $v_0 = b_0[1],\ldots,b_0[m]$. For all $k > 0$, let $t_k = t(P_b(v_{k-1}))$. If $t_k \leq n$, define $P_a(a_1,\ldots,a_k) = P_a(a_1,\ldots,a_{k-1})$ and let $v_k = v_{k-1}, a_k[t_k]$. If, on the other hand, $t_k > n$, then let $P_b(v_{k-1}) = (d_k,t_k)$ for some $d_k \in D$ and for all $1 \leq i \leq m$, if $i = t_k$ then define $P_a(a_1,\ldots,a_k)[i] = d_k$ and otherwise ($i \neq t_k$) define $P_a(a_1,\ldots,a_k)[i] = P_a(a_1,\ldots,a_{k-1})[i]$. Also, if $t_k > n$ then let $v_k = v_{k-1}, d_k$.

We now show that all asynchronous interactions of P_a, for all schedules,

satisfy $\varphi(X;Y)$, implying that $\varphi(X;Y)$ is asynchronously realized by P_a. Let (R,W) be a schedule, and let $\sigma = (a_0,b_0),(a_1,b_1),\ldots$ be an asynchronous interaction of P_a for this schedule. Denote $R = r_1^1,\ldots,r_1^n,r_2^1,\ldots,r_2^n,\ldots$ and $W = w_1^1,\ldots,w_1^m,w_2^1,\ldots,w_2^m,\ldots$.

We note that σ is also a by-demand asynchronous interaction of P_b, where the $I\backslash O$ points are restricted to either one reading point or one writing point from $r_i^1,\ldots,r_i^n,w_i^1,\ldots,w_i^m$ for every i. More formally, let $C = c_0,c_1,\ldots$ be the $I\backslash O$ sequence that produces σ as a by-demand asynchronous interaction of P_b. We set $c_0 = 0$ and for all $i > 0$ let $t(P_b(v(P_b,\sigma,C,i-1))) = j$, and if $j \leq n$ then $c_i = r_i^j$ and if $j > n$ then $c_i = w_i^j$. Note that $t(\cdot)$ and $v(\cdot)$ above are well defined as $v(P_b,\sigma,C,i-1)$ requires only the elements of C up to the $i-1$-th element.

As σ satisfies $\varphi(X;Y)$ (by correctness of P_b), we are done.

$\Rightarrow$ Let $\varphi(X;Y)$ be asynchronously realized by the asynchronous program $P_a : (D^X)^* \mapsto D^Y$. We define the by-demand asynchronous program $P_b : D^* \mapsto \{1,\ldots,n\} \cup (D \times \{n+1,\ldots,n+m\})$ as shown below. Intuitively, P_b reads and writes the appropriate variables in a cyclical order, mimicking the behavior of P_a.

For a sequence $\tau = (a_1,b_1),\ldots,(a_k,b_k) \in (D^X \times D^Y)^*$ we define the *unwinding* $q(\tau)$ as the sequence of individual values for individual variables that appear in the sequence τ. We define concurrently P_a and the function $q : (D^X \cup D^Y)^* \mapsto D^+$. Let $P_a(\epsilon) = b_0$. Then, for all $0 \leq i < m$ let $d_i = b_0[i]$ and define $P_b(d_1,\ldots,d_i) = (d_{i+1},(n+i+1))$. Also define $P_b(d_1,\ldots,d_m) = 1$. Define $q(\epsilon) = d_1,\ldots,d_m$. For all $\tau \in (D^X \cup D^Y)^*$,

$a \in D^X$ and $0 < i \leq n$ let $a[i] = d_i$ and define $q(\tau, a) = q(\tau), d_1, \ldots, d_n$. In addition, for all $\tau \in (D^X \cup D^Y)^*$, $a \in D^X$, $b \in D^Y$ and $0 < i \leq m$ let $b[i] = d_i$ and define $q(\tau, a, b) = q(\tau, a), d_1, \ldots, d_m$. In general, consider $k > 0$ and let $P_a(a_1, \ldots, a_k) = b_k$, where for all $0 < i \leq n$ we have $a_k[i] = d_i$ and for all $n < i \leq n+m$ we have $b_k[i-n] = d_i$. Then, for every $0 < i < n$ we set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_i) = i+1$, for every $n \leq i < n+m$ we set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_i) = (d_{i+1}, (i+1))$, and we finally set $P_b(q\Big((a_1, b_1), \ldots, (a_{k-1}, b_{k-1})\Big), d_1, \ldots, d_{n+m}) = 1$.

We now show that all by-demand asynchronous interactions of P_b, for all $I\backslash O$-sequences, satisfy $\varphi(X;Y)$, implying that $\varphi(X;Y)$ is by-demand asynchronously realized by P_b. Let C be an $I\backslash O$ sequences, and let the sequence $\sigma = (a_0, b_0), (a_1, b_1), \ldots$ be a by-demand asynchronous interaction of P_b for this $I\backslash O$ sequences. Denote $C = c_0, c_1, c_2, \ldots$, where $c_0 = 0$.

We define a schedule (R, W), where $R = r_1^1, \ldots, r_1^n, r_2^1, \ldots, r_2^n, \ldots$ and $W = w_1^1, \ldots, w_1^m, w_2^1, \ldots, w_2^m, \ldots$, as follows. For all $i > 0$, consider the unique j and $0 < k \leq n+m$ such that $c_i = c_{j \cdot (n+m)+k}$. If $k \leq n$ then define r_{j+1}^k, and otherwise (if $n < k$) define w_{j+1}^{k-n}. This defines (R, W) completely. We note that σ is also an asynchronous interaction of P_a for (R, W).

As σ satisfies $\varphi(X;Y)$ (by correctness of P_a), we are done.

$\square$

From Theorem 3.4 it is clear that, despite the appearance of greater freedom allowed by by-demand asynchronous programs in comparison with 'regular' (or round-robin) asynchronous programs, the two are in fact equivalent for purposes of synthesis. That means that one loses no generality by assuming a round-robin,

cyclical, pattern or $I\backslash O$ operations when considering asynchronous interactions. For completeness, however, we present in Subsection 3.4.2 a reduction that is equivalent to the generalized Rosner reduction and which uses by-demand asynchronous programs, effectively reducing the problem of by-demand asynchronous synthesis to that of synchronous synthesis.

3.4.2 A Modified Generalized Rosner Reduction

Inspired by the Rosner reduction from Subsection 3.2.4, we propose a family of translations (one translation for each pair (n, m)) called the *by-demand generalized Rosner reduction*. We translate a specification $\varphi(X; Y)$ into a QPTL specification $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ that has an additional Boolean input variable c and an additional output variable h that ranges over $\{1, \ldots, (n+m)\}$. The new variable c is called the *Boolean $I\backslash O$ variable*, and h is called the *$I\backslash O$-selector variable*.

Similar to the role of r in the previous reduction, a change in the value of c indicates an $I\backslash O$-point. The value of h indicates the choice of which variable to read\write. Values $1, \ldots, n$ indicate reading and values $(n+1), \ldots, (n+m)$ indicate writing. We set a new value for h right after a read $I\backslash O$ write. Thus, the system immediately commits to the next variable it is going to access. As h is treated like all other outputs, the system cannot change its value when c does not change. This corresponds to no new information being gained by the system as long as c does not change.

In this section, we use our notations $\mathit{unchanged}(x)$ to indicate a state where variable x did not change its value, and *first* to indicate a state that is the first one in the computation.

The kernel formula $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\}$ is defined as follows.

$$\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\}) = \gamma(c) \rightarrow \delta^{n,m}(X \cup \{c\}; Y \cup \{h\}),$$

where

$$\gamma(c) = \bar{c} \wedge \Box \Diamond c \wedge \Box \Diamond \bar{c}$$

and $\delta^{n,m}(X \cup \{c\}; Y \cup \{h\})$ is given by

$$\begin{pmatrix} [(c \leftrightarrow \ominus c) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(h) & \wedge \\ \varphi(X; Y) & \wedge \\ \bigwedge_{i=1}^{m} \left\{ \left[\left[(c \leftrightarrow \ominus c) \vee \ominus [h \neq (i+n)] \right] \wedge \neg \mathit{first} \right] \Rightarrow \mathit{unchanged}(y_i) \right\} & \wedge \\ (\forall^{\approx} \tilde{X}). \left[\bigwedge_{i=1}^{n} [[\neg (c \leftrightarrow \ominus c) \wedge \ominus (h = i)] \Rightarrow (x_i = \tilde{x}_i)] \right] \rightarrow \varphi(\tilde{X}; Y) & \end{pmatrix}.$$

The initial value of the $I \backslash O$-selector variable h is selected nondeterministically.

Similar to the role of the scheduling variable in $\mathcal{X}^{n,m}$, the variables c and h in $\mathcal{Y}^{n,m}$ make explicit the decisions of the environment when to have an $I \backslash O$ point for the system (c), and which variable to read\write (h).

Theorem 3.5. *Let $\varphi(X; Y)$ be a* LTL *specification where $|X| = n$ and $|Y| = m$, and let its kernel formulae be $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ and $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, derived by the by-demand generalized Rosner reduction, and by the generalized Rosner reduction, respectively.*

The specification $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$ is synchronously realizable **iff** *the specification $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable. Furthermore, given a program $P_{\mathcal{Y}}$ that synchronously realizes $\mathcal{Y}^{n,m}$, it can be converted to a program $P_{\mathcal{X}}$ that synchronously realizes $\mathcal{X}^{n,m}$ in time linear in the number of transitions of the*

LTS that induces $P_{\mathcal{X}}$, and vice versa.

Proof: We shall prove both directions:

$\Leftarrow$ Having that $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable, means that there exists a program $P_{\mathcal{X}}$ that synchronously realizes it. We describe the construction of another program, $P_{\mathcal{Y}}$, that synchronously realizes $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, effectively proving that it is synchronously realizable.

To do this, we need to provide a function that corresponds to $P_{\mathcal{Y}}$, and that generates assignments to $Y \cup \{h\}$, given finite histories of assignments to $X \cup \{c\}$. We then need to prove that all synchronous interaction of this program which we construct, satisfy $\mathcal{Y}^{n,m}$.

Let $\sigma_{X,c} = \sigma_0, \sigma_1, \ldots, \sigma_k$ be a finite history of assignments to $X \cup \{c\}$ for $k \geq 0$, such that for $0 \leq i \leq k$, $\sigma_i = (X_i, c_i)$ where X_i is an assignment to X and c_i is an assignment to c. We construct the sequence of assignments to r, $\eta_r = r_0, r_1, \ldots, r_k$, in the following way: $r_0 = n + m$. For $k \geq i > 0$, if $c_i \leftrightarrow \ominus c_{i-1}$ then $r_i = r_{i-1}$, otherwise $r_i = r_{i-1} \oplus_{n+m} 1$.

Let an assignment to $Y \cup \{h\}$ be a pair (Y_i, h_i) where Y_i is an assignment to Y and h_i is an assignment to h. Similarly, an assignment to $X \cup \{r\}$ is a pair (X_i, r_i) where X_i is an assignment to X and r_i is an assignment to r.

We define

$$P_{\mathcal{Y}}(\sigma_{X,c}) = \left(\underbrace{P_{\mathcal{X}}\Big((X_0, r_0), (X_1, r_1), \ldots, (X_k, r_k)\Big)}_{Y_k}, \underbrace{r_k \oplus_{n+m} 1}_{h_k} \right)$$

where Y_k is an assignment to Y and h_k is an assignment to h.

It is not difficult to see why any computations that would be based on a synchronous interaction with $P_{\mathcal{Y}}$ would satisfy $\mathcal{Y}^{n,m}$. If in a particular computation $\mu_{X,c}$ the $I\backslash O$ variable c does not change its value infinitely often, then $\mathcal{Y}^{n,m}$ is trivially satisfied. Otherwise, by the way we construct μ_r (a computation for $\{r\}$), $\mu_r, 0 \models \alpha^{n,m}(r)$, and we know that $\mu_{X,r,Y}$ (using the Y values that we output from $P_{\mathcal{X}}$) satisfies all three conjuncts of $\beta^{n,m}(X \cup \{r\}; Y)$. The first one, $\varphi(X;Y)$, appears also in $\mathcal{Y}^{n,m}$. The second one turns out to be essentially identical to the third conjunct in $\delta^{n,m}$, since h is identical to r at all $I\backslash O$ points, and since c and r change their values always together. The last conjuncts in both $\beta^{n,m}$ and $\delta^{n,m}$ are essentially identical for the same reasons. The remaining conjunct in $\delta^{n,m}$, the first one, is also satisfied by h starting with the value $r_0 \oplus_{n+m} 1 = (n+m) \oplus_{n+m} 1 = 1$, and by $[(c \leftrightarrow \ominus\, c) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(h)$ holding due to the fact that we change r **iff** c changes, and we change h **iff** r changes.

$\Rightarrow$ In this direction we know that there exists a program $P_{\mathcal{Y}}$ that synchronously realizes $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, and we construct a program $P_{\mathcal{X}}$ that synchronously realizes $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. We use similar notations of assignments and histories as in the other direction.

Let $\sigma^k_{X,r} = \sigma_0, \sigma_1, \ldots, \sigma_k$ be a prefix for $k \geq 0$ of a computation over the variables in $X \cup \{r\}$ (we freely use similar notations for other sets of variables). If $r_0 \neq (n+m)$, or if there exists some index $i > 0$ such that $(r_i \neq r_{i-1}) \wedge [r_i \neq (r_{i-1} \oplus_{n+m} 1)]$ holds, then for all $j \geq i$ (for all $j \geq 0$ if $r_0 \neq (n+m)$) we define $Y_j = Y_{rand}$ for some arbitrary $Y_{rand} \in D^Y$. From this point onwards in the construction we assume that this is not the case, and that r updates as indicated by $\alpha^{n,m}(r)$. We construct the prefixes $\sigma^k_{c,Y,h}$, for all k, inductively

(using the initial value of h, h_0, which is deterministically selected by $P_{\mathcal{Y}}$):

- Let i_1 be the minimal index such that r_{i_1} changes its value to h_0 (so that $r_{i_1} = h_0$). We know that there exists $i_1 > 0$ since we assume that r changes cyclically infinitely often. Let $\sigma_c^{i_1} = c_0, \ldots, c_{i_1}$, where $\overline{c_0}$ holds, and for $0 < j < i_1$ $\overline{c_j}$ holds. c_{i_1} holds as well ($c_{i_1} = \mathrm{T}$). For $0 \leq j \leq i_1$ $P_{\mathcal{Y}}((X_0, c_0), \ldots, (X_j, c_j)) = (Y_j, h_j)$.
- Let i_t be the minimal index that is greater than i_{t-1} such that r_{i_t} changes its value to $h_{i_{t-1}}$ (so that $r_{i_t} = h_{i_{t-1}}$). We know that there exists such i_t. Let $\sigma_c^{i_t} = \sigma_c^{i_{t-1}}, c_{i_{t-1}+1}, \ldots, c_{i_t}$, where for $i_{t-1}+1 \leq j < i_t$ $c_j \leftrightarrow c_{i_{t-1}}$ holds. $\neg(c_{i_t} \leftrightarrow c_{i_{t-1}})$ holds as well. For $i_{t-1} < j \leq i_t$ $P_{\mathcal{Y}}((X_0, c_0), \ldots, (X_j, c_j)) = (Y_j, h_j)$.

Using the construction and definitions described above, we define

$$P_{\mathcal{X}}(\sigma_{X,r}^k) = Y_k$$

where Y_k is an assignment to Y.

It is not difficult to see why any computations that would be based on a synchronous interaction with $P_{\mathcal{X}}$ would satisfy $\mathcal{X}^{n,m}$. If in a particular computation $\mu_{X,r}$ $\mu_r, 0 \not\models \alpha^{n,m}(r)$ then $\mathcal{X}^{n,m}$ is trivially satisfied with Y_{rand}. Otherwise, we can definitely construct σ_c^k for all k (since r changes cyclically infinitely often, and therefore admits all of its domain values infinitely often). Since in each iteration of σ_c^k construction we have exactly one change of c value, then $\mu_c, 0 \models \gamma(c)$. Since we constructed $\mu_{Y,h}$ using $P_{\mathcal{Y}}$, we get that $\mu_{X,c,Y,h}$ satisfies all four conjuncts of $\delta^{n,m}(X \cup \{c\}; Y \cup \{h\})$. The second one, $\varphi(X; Y)$, appears also in $\mathcal{X}^{n,m}$. Noticing that the set of states in $\mu_{X,c,r,Y,h}$

that satisfy $(c \leftrightarrow \ominus c) \vee \ominus[h \neq (i+n)]$ is a super-set of the states that satisfy $\neg write_n(i)$ (for all $i \in \{1, \ldots, m\}$), we get that the satisfaction of the third conjunct in $\delta^{n,m}$ guarantees the satisfaction of the second conjunct in $\beta^{n,m}$. Finally, since the set of states in $\mu_{X,c,r,Y,h}$ that satisfy $\neg(c \leftrightarrow \ominus c) \wedge \ominus(h = i)$ is a sub-set of the states that satisfy $read(i)$ (again, for all $i \in \{1, \ldots, m\}$), we get that the satisfaction of the last conjunct in $\delta^{n,m}$ guarantees the satisfaction of the last conjunct in $\beta^{n,m}$ (since there are fewer reading points in $\mathcal{Y}^{n,m}$, then $\varphi(\tilde{X}; Y)$ must hold for a large set of computations $\tilde{X}$, including all of those that are 'allowed' by the clause $\varphi(\tilde{X}; Y)$ in $\mathcal{X}^{n,m}$).

□

Theorem 3.6. *Given a specification $\varphi(X; Y)$ ($|X| = n$, and $|Y| = m$), the following conditions are equivalent:*

1. *$\varphi(X; Y)$ is by-demand asynchronously realizable.*
2. *$\varphi(X; Y)$ is asynchronously realizable.*
3. *The kernel formula $\mathcal{Y}^{n,m}(X \cup \{c\}; Y \cup \{h\})$, which is derived from $\varphi(X; Y)$ using the by-demand generalized Ronser reduction, is synchronously realizable.*
4. *The kernel formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$, which is derived from $\varphi(X; Y)$ using the generalized Ronser reduction, is synchronously realizable.*

Furthermore, given a program P that realizes one of these specifications, it can be converted to a program that realizes any of the other in time linear in the number of transitions of the LTS/ILTS that induces P.

Proof: This is a direct result of Theorem 3.3, Theorem 3.4, and Theorem 3.5. □

Theorem 3.6 finally confirms that both $\mathcal{Y}^{n,m}$ and $\mathcal{X}^{n,m}$ may be freely used to test for any type of asynchronous realizability of $\varphi(X;Y)$, as well as for synthesis. From this point onward we consider only round-robin asynchronous realizability and the reduction from $\varphi(X;Y)$ to $\mathcal{X}^{n,m}$.

3.5 Proving Unrealizability of a Specification

In this section we show how an over-approximation of $\mathcal{X}^{n,m}$ can effectively prove that a given specification is asynchronously unrealizable.

3.5.1 Over-Approximating the Kernel Formula

Fix an LTL specification $\varphi(X;Y) = \mathit{Imp}(\varphi_e, \varphi_s)$. Let $X = \{x_1, \dots, x_n\}$, $Y = \{y_1, \dots, y_m\}$, and let r be a scheduling variable ranging over $\{1, \dots, (n+m)\}$. Let $\tilde{X} = \{\tilde{x} | x \in X\}$. We assume that $r \notin X \cup Y$ and that $\tilde{X} \cap (X \cup Y) = \emptyset$.

In this section, we use our notations $\mathit{write}_n(i)$ to indicate a state that is a writing point for the i'th output, $\mathit{read}(i)$ to indicate a state that is a reading point for the i'th input, $\mathit{unchanged}(x)$ to indicate a state where variable x did not change its value, and *first* to indicate a state that is the first one in the computation.

As explained in Section 3.3, the generalized Rosner reduction, although offering a complete solution to the asynchronous synthesis problem, often translates into a prohibitively costly to synthesize specification due to the universal quantification in the clause $\beta_3^{n,m}$. If we wish to use 'effective' algorithms for synthesis based on this reduction, we must find a way to avoid the size increase caused by $\beta_3^{n,m}$.

Recall the generalized Rosner reduction formula $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. As noted in Subsection 3.2.4, the first conjunct in $\beta^{n,m}$ – $\varphi(X;Y)$ – is a redundant one that was left in the Rosner reduction and in its generalizations only for clarity purposes (this conjunct follows from $\beta_3^{n,m}$). Therefore, in this section we allow ourselves to remove $\varphi(X;Y)$ from $\beta^{n,m}$, leaving us with

$$\tilde{\mathcal{X}}^{n,m}(X \cup \{r\}; Y) = \alpha^{n,m}(r) \rightarrow \tilde{\beta}^{n,m}(X \cup \{r\}; Y)$$

where $\tilde{\beta}^{n,m}(X \cup \{r\}; Y)$ is given by

$$\begin{pmatrix} \bigwedge_{i=1}^{m} \Big[[\neg write_n(i) \wedge \neg first] \Rrightarrow unchanged(y_i) \Big] & \wedge \\ (\forall^{\approx} \tilde{X}). \Big[\bigwedge_{i=1}^{n} [read(i) \Rrightarrow (x_i = \tilde{x}_i)] \Big] \rightarrow \varphi(\tilde{X}; Y) & \end{pmatrix}.$$

We know that $\tilde{\mathcal{X}}^{n,m} \leftrightarrow \mathcal{X}^{n,m}$, and we may use them interchangeably. We still use $\beta_3^{n,m}$ as a name for the last conjunct of $\beta^{n,m}$ and of $\tilde{\beta}^{n,m}$

With this in mind, we define an *over-approximating formula* for $\mathcal{X}^{n,m}$:

$$\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y) = \alpha^{n,m}(r) \rightarrow \beta_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$$

where $\beta_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$ is given by

$$\begin{pmatrix} \bigwedge_{i=1}^{m} \Big[[\neg write_n(i) \wedge \neg first] \Rrightarrow unchanged(y_i) \Big] & \wedge \\ \Big[\bigwedge_{i=1}^{n} [read(i) \Rrightarrow (x_i = \tilde{x}_i)] \Big] \rightarrow \varphi(\tilde{X}; Y) & \end{pmatrix}.$$

Note that $\mathcal{X}_{\downarrow}^{n,m}$ is almost identical to $\mathcal{X}^{n,m}$ (after removing its redundant clause),

except that the second clause in $\beta_{\downarrow}^{n,m}$ has no quantification (over $\tilde{X}$), eliminating the source of trouble in $\beta_3^{n,m}$. In effect, this amounts to adding $\tilde{X}$ to the set of input variables. In fact, if $\varphi(X;Y)$ has a $GR(1)$ winning condition, then it could be easily shown by propositional arguments[8] that $\mathcal{X}_{\downarrow}^{n,m}$ has a $GR(1)$ winning condition as well. If we can show that it is possible to deduce of synchronous unrealizability of $\mathcal{X}^{n,m}$ from that of $\mathcal{X}_{\downarrow}^{n,m}$, then we could use the effective algorithm of [PPS06] on $\mathcal{X}_{\downarrow}^{n,m}$.

The main observation relating $\mathcal{X}^{n,m}$ to $\mathcal{X}_{\downarrow}^{n,m}$ is the following theorem:

Theorem 3.7. *For a specification $\varphi(X;Y)$ where $|X| = n$ and $|Y| = m$, and for a scheduling variable r ranging over $\{1, \ldots, (n+m)\}$, the following holds: If $\mathcal{X}^{n,m}(X \cup \{r\};Y)$ is synchronously realizable, then $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\};Y)$ is synchronously realizable. (Here, $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\};Y)$ and $\mathcal{X}^{n,m}(X \cup \{r\};Y)$ are the appropriate kernel formulae derived for $\varphi(X;Y)$.)*

Proof: Let P_s be a program that synchronously realizes $\mathcal{X}^{n,m}(X \cup \{r\};Y)$. We shall prove that P_s also synchronously realizes $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\};Y)$.

Let σ be a computation that is induced by $\mathcal{X}^{n,m}$, so that $\sigma, 0 \models \mathcal{X}^{n,m}$. We would like to show that $\sigma, 0 \models \mathcal{X}_{\downarrow}^{n,m}$.

If $\sigma, 0 \not\models \alpha^{n,m}$, then, trivially, $\sigma, 0 \models \mathcal{X}_{\downarrow}^{n,m}$. Otherwise, we know that σ satisfies also the first conjunct in $\tilde{\beta}^{n,m}$ which appears in $\beta_{\downarrow}^{n,m}$. To prove that $\sigma, 0 \models \mathcal{X}_{\downarrow}^{n,m}$, we are left with proving that σ satisfies the last (second) conjunct in $\beta_{\downarrow}^{n,m}$.

We also know, however, that σ satisfies the last conjunct in $\beta^{n,m}$: $\sigma, 0 \models (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} [read(i) \Rightarrow\!\!\succ (x_i = \tilde{x}_i)] \rightarrow \varphi(\tilde{X};Y)$. Since this means that the implication that appears in this conjunct would be satisfied by any $\tilde{X}$-variant σ'' of

[8] Roughly speaking, all elements of $\mathcal{X}_{\downarrow}^{n,m}$ could be 'absorbed' into $\varphi(\tilde{X};Y)$ without increasing the formula's complexity in terms of the temporal hierarchy.

any stuttering variant σ' of σ, we only weaken this statement by writing that $\sigma, 0 \models \bigwedge_{i=1}^{n} [read(i) \Rightarrow (x_i = \tilde{x}_i)] \rightarrow \varphi(\tilde{X}; Y)$. Since this is a weakening transition (claiming satisfiability by σ only), it is correct. This is, however, exactly the last conjunct in $\beta_{\downarrow}^{n,m}$, and the proof is complete. □

An important result of Theorem 3.7 is the following:

Theorem 3.8. *For a specification $\varphi(X; Y)$ where $|X| = n$ and $|Y| = m$, and for a scheduling variable r ranging over $\{1, \ldots, (n + m)\}$, the following holds: If $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$ is synchronously unrealizable, then $\varphi(X; Y)$ is asynchronously unrealizable. (Here, $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$ is the appropriate kernel formulae derived for $\varphi(X; Y)$.)*

Proof: This is a direct result of Theorem 3.1, Theorem 3.3 and Theorem 3.7. □

Theorem 3.8 provides us with the framework for an effective way to test whether specifications with $GR(1)$ winning conditions are asynchronously unrealizable, as desired. This is what justifies referring to $\mathcal{X}_{\downarrow}^{n,m}(X \cup \tilde{X} \cup \{r\}; Y)$ as an **over-approximation** (equivalently, weakening) of $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. Indeed, the effective algorithm of [PPS06] could be used with $\mathcal{X}_{\downarrow}^{n,m}$ to test whether its underlying specification $\varphi(X; Y)$ is asynchronously unrealizable. The time complexity of using this algorithm (for specifications with $GR(1)$ winning conditions) is $O(N^3 \cdot m \cdot n)$, where N is the state space of the specification, and m and n are the number of liveness conjuncts of the environment and system's specifications, respectively.

Caveat: There exist specifications which the effective algorithm we propose – as described in [PPS06] – declare synchronously unrealizable, while they are in fact realizable. This may lead to a false classification of $\mathcal{X}_{\downarrow}^{n,m}$ formulae as synchronously

unrealizable, and therefore to a 'false-negative' of declaring the underlying specification φ asynchronously unrealizable. Such specifications with $GR(1)$ winning conditions are characterized in Chapter 2, where an efficient method is described for identifying them, as well as a method for avoiding false-negatives given such specifications.

Note that while the methods proposed in this section are sound, they are not complete, in the sense that a specification $\varphi(X;Y)$ may be asynchronously unrealizable but the derived synchronous approximation $\mathcal{X}_{\downarrow}^{n,m}$ may be synchronously realizable.

3.5.2 Applying the Unrealizability Test

In this subsection we illustrate the application of the effective unrealizability test based on Theorem 3.8.

We start with the 'copy' specification $\varphi_1(x;y) : \square(x \leftrightarrow y)$ which we considered in the introduction (both x and y are Booleans, and for clarity, we do not specify here an initial condition - $\overline{x} \wedge \overline{y}$). In our discussion there we claimed that this specification is asynchronously unrealizable but stated this fact with no proof. Now, we have an adequate tool for proving that this specification is indeed asynchronously unrealizable. Deriving the kernel formula $\mathcal{X}_{\downarrow}^{1,1}(x, \tilde{x}, r; y)$ for $\varphi_1(x;y)$, we obtain the specification $\mathcal{X}_{\downarrow}\langle\varphi_1\rangle = \alpha^{1,1}(r) \rightarrow \beta_{\downarrow}\langle\varphi_1\rangle$, where $\beta_{\downarrow}\langle\varphi_1\rangle(x, \tilde{x}, r; y)$ is given by

$$\begin{pmatrix} (((r \neq 2) \vee \ominus(r \neq 1)) \wedge \neg \mathit{first}) \Rightarrow \mathit{unchanged}(y) & \wedge \\ (((r = 1) \wedge \ominus(r \neq 1)) \Rightarrow (x \leftrightarrow \tilde{x})) \rightarrow \square(\tilde{x} \leftrightarrow y) & \end{pmatrix}.$$

We proceed to show that there can be no synchronous program that satisfies (by controlling y) $\mathcal{X}_{\downarrow}\langle\varphi_1\rangle(x, \tilde{x}, r; y)$ for all choices of x, $\tilde{x}$, and r. Assume the opposite,

and consider a computation $\sigma = a_0, a_1, \ldots$, such that $\overline{x}$ holds at all states, and $\tilde{x}$ and $r = 2$ hold at a state a_j **iff** j is odd. It follows that all even indexed states are reading points, and $x = \tilde{x}$ at all of these states. Consequently, and since $\sigma, 0 \models \alpha^{1,1}(r)$, we should have $\sigma, 0 \models \square(y \leftrightarrow x)$ and $\sigma, 0 \models \square(\tilde{x} \leftrightarrow y)$. However, this implies that $x \leftrightarrow \tilde{x}$ should hold at all states, which is false because x and $\tilde{x}$ differ at all odd-indexed states.

We conclude that the specification $\varphi_1(x; y) : \square(x \leftrightarrow y)$ is asynchronously unrealizable. Indeed, when checking synchronous realizability of the kernel formula $\mathcal{X}^{1,1}_{\downarrow}(x, \tilde{x}, r; y)$ for $\varphi_1(x; y)$ using the algorithm of [PPS06], we get that it is unrealizable.

Assume that we are not ready to give up and would like to develop an asynchronous system that captures some of the essential behavior of a copying module. An informal description of such a behavior can include the following requirements:

1. Whenever x rises to 1, then sometimes later y should rise to 1.
2. Whenever x drops to 0, then sometimes later y should drop to 0.
3. Variable y should not rise to 1, unless sometimes before x was 1.
4. Variable y should not drop to 0, unless sometimes before x was 0.

A temporal formula that captures these four requirements may be given by the following specification:

$$\varphi_2(x; y) : \begin{pmatrix} (x \Rightarrow \diamondsuit\, y) \wedge (\overline{x} \Rightarrow \diamondsuit\, \overline{y}) \wedge \\ (y \Rightarrow y \,\mathcal{S}\, \overline{y} \,\mathcal{S}\, x) \wedge \bigcirc(\overline{y} \Rightarrow \overline{y} \,\mathcal{B}\, y \,\mathcal{S}\, \overline{x}) \end{pmatrix}$$

As before, both x and y are Booleans, and for clarity, we do not specify here an initial condition - $\overline{x} \wedge \overline{y}$. The past formula $y \Rightarrow y \,\mathcal{S}\, \overline{y} \,\mathcal{S}\, x$ states that if currently

y holds then this state was preceded by an interval in which y continuously held, preceded by an interval in which $\overline{y}$ continuously held, preceded by a state at which x held. The formula $\bigcirc(\overline{y} \Rightarrow \overline{y}\, \mathcal{B}\, y\, \mathcal{S}\, \overline{x})$ states that, starting at the second state, if currently $\overline{y}$ holds, then this state is preceded by an interval in which $\overline{y}$ continuously held, and which either extends to the beginning of the computation or is preceded by an interval in which y continuously held and which is preceded by a state at which $\overline{x}$ held.

We will now apply the unrealizability test to check whether $\varphi_2(x;y)$ is also asynchronously unrealizable. In order to conclude that this is the case, we have to find a computation $\sigma = a_0, a_1, \ldots$ in which x and $\tilde{x}$ agree infinitely often (reading points), and where, regardless of y values, one of the following must hold: $\sigma, 0 \not\models \varphi_2(x;y)$ or $\sigma, 0 \not\models \varphi_2(\tilde{x};y)$. Assume that $\overline{x}$ holds at all states, and let $\tilde{x}$ hold at state a_j **iff** j is odd. Thus, we can take all even-indexed states to be the reading points, and $x = \tilde{x}$ at all of these states. $\sigma, 0 \models \varphi_2(x;y)$ implies that $\overline{y}$ holds at all states. This is since any occurrence of y at some state implies, by $y \Rightarrow y\, \mathcal{S}\, \overline{y}\, \mathcal{S}\, x$, that x holds at some earlier state, which never is the case. However, in this case, the fact that $\tilde{x}$ holds at state a_1 entails that $\sigma, 0 \not\models \varphi_2(\tilde{x};y)$ because it violates the requirement $\tilde{x} \Rightarrow \Diamond\, y$, which is part of $\varphi_2(\tilde{x};y)$. We conclude that $\varphi_2(x;y)$ is also asynchronously unrealizable. Again, when checking synchronous realizability of the kernel formula $\mathcal{X}_{\downarrow}^{1,1}(x,\tilde{x},r;y)$ for $\varphi_2(x;y)$ using the algorithm of [PPS06], we get that it is unrealizable.

How can we weaken $\varphi_2(x;y)$ into a specification that stands a better chance of being asynchronously realizable? Obviously, the weakness of the specification $\varphi_2(x;y)$ is that it allows the environment to modify x too quickly without waiting for an evidence that the system has noticed the most recent change. We can correct

this drawback by allowing the environment to modify x only at points in which $x \leftrightarrow y$ (that is, after the system had enough time to respond to a change of x). For example, we can suggest the following 'response' specification:

$$\varphi_3(x;y) : [\neg(x \leftrightarrow y) \Rrightarrow (x \leftrightarrow \bigcirc x)] \rightarrow \begin{pmatrix} x \Rrightarrow \Diamond y & \wedge \\ \overline{x} \Rrightarrow \Diamond \overline{y} & \wedge \\ y \Rrightarrow y \, \mathcal{S} \, \overline{y} \, \mathcal{S} \, x & \wedge \\ \bigcirc(\overline{y} \Rrightarrow \overline{y} \, \mathcal{B} \, y \, \mathcal{S} \, \overline{x}) & \end{pmatrix}$$

As before, both x and y are Booleans, and for clarity, we do not specify here an initial condition - $\overline{x} \wedge \overline{y}$. Applying the unrealizability test to $\varphi_3(x;y)$, we find that its corresponding kernel formula $\mathcal{X}^{1,1}_{\downarrow}(x, \tilde{x}, r; y)$ is synchronously realizable. However, we cannot infer any conclusions from this, since Theorem 3.8 offers only conclusions in the unrealizable case. In the next section, we will consider methods that can lead to effective realizability and apply them to the specification $\varphi_3(x;y)$.

3.6 Proving Realizability of a Specification, and Synthesis

As mentioned, the formula $\mathcal{X}^{n,m}$ has not led to a practical solution for asynchronous synthesis. Here, we are interested in finding cases where synthesis can be applied in practice by circumventing the complexity of determinization and solving of complex parity / Rabin games. However, even when starting from a relatively simple formula $\varphi(X;Y)$, the third conjunct in $\beta^{n,m}$, which includes the operator $\forall^{\approx}$, forces us to use the complex synthesis algorithm in [PR89a]. Here, we concentrate on specifications with winning conditions that fall into the class of $GR(1)$ formulae, as defined in Subsection 3.2.3. For such formulae we find cases in which we can bypass the use of the third conjunct in the definition of $\beta^{n,m}$. If successful we

can apply simpler synthesis algorithms, e.g., those in [PPS06]. Given the formula $\varphi(X;Y)$, of the required form, we aim to find a strengthening of it that will allow to use a simpler reduction to synchronous synthesis.

3.6.1 Under-Approximating the Kernel Formula

We fix a specification $\varphi(X;Y) = \mathit{Imp}(\varphi_e, \varphi_s)$ with a $GR(1)$ winning condition, where $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and $\varphi_e = \langle I_{\varphi_e}, S_{\varphi_e}, L_{\varphi_e}\rangle$. Let r be a scheduling variable ranging over $\{1, \ldots, (n+m)\}$ and let $\tilde{X} = \{\tilde{x} | x \in X\}$. We define the set of *declared output variables* as $\tilde{Y} = \{\tilde{y} | y \in Y\}$. We assume that $r \notin X$, $\tilde{X} \cap Y = \emptyset$, and that $\tilde{Y} \cap X = \emptyset$.

In this section, we use our notations $\mathit{write}_n(i)$ to indicate a state that is a writing point for the i'th output, $\mathit{read}(i)$ to indicate a state that is a reading point for the i'th input, $\mathit{unchanged}(x)$ to indicate a state where variable x did not change its value, and *first* to indicate a state that is the first one in the computation.

We define another formula $\psi(X \cup \{r\}; Y)$ with a $GR(1)$ winning condition, that strengthens $\varphi(X;Y)$. The role of this formula, called $\psi(X \cup \{r\}; Y)$ is to allow us to form a kernel formula where we simply eliminate the 'problematic' clause $\beta_3^{n,m}$, and that also, somehow (given that it is synchronously realizable), leads to the synthesis of a programs that satisfies φ. So this new specification that we want, ψ, must be strong enough to both imply, in some sense, φ, and to lead to the formation of a synchronously realizable kernel formula from which we could extract a program for the underlying φ. The way to construct candidates for such strengthening is a heuristic that is presented in Subsection 3.6.2. The algorithm for using such a formula ψ, and the proof that it works, are presented below and in Subsection 3.6.2.

$\psi(X \cup \{r\}; Y)$ includes r in its outputs since it is solely designed to be 'fitted' into a kernel formula from which a program for $\varphi(X;Y)$ would be extracted. Since ψ should never be synthesized in itself, and since the scheduling variable is present in the kernel formula, we allow ψ the maximal degree of freedom by allowing it to refer to the scheduling variable. Beyond this degree of freedom, there is some intuition to this choice: Since the clause that we wish to eliminate, $\beta_3^{n,m}$, is dealing with the requirement that $\varphi(X;Y)$ depends only on inputs at reading points, one possible way to make it redundant would be to make sure, explicitly in ψ, that system's transitions only depend on inputs in reading points. To do this, ψ must refer to the scheduling variable. Indeed, this is exactly the approach that we take in our heuristic to construct candidates for such ψ formulae.

Definition 3.3 (asynchronous strengthening)**.** *A specification* $\psi(X \cup \{r\}; Y) = \mathit{Imp}(\psi_e, \psi_s)$ *with a* $GR(1)$ *winning condition, where* $\psi_e = \langle I_{\psi_e}, S_{\psi_e}, L_{\psi_e}\rangle$*, is an* **asynchronous strengthening** *of* $\varphi(X;Y)$ *if the following conditions hold:*

1. $I_{\psi_e} = I_{\varphi_e}$.
2. $S_{\psi_e} = S_{\varphi_e}$.
3. *The following implication is valid*

$$\begin{pmatrix} \alpha^{n,m}(r) & \wedge \\ I_{\psi_e} \wedge \square\, S_{\psi_e} & \wedge \\ \psi(X \cup \{r\}; Y) & \wedge \\ \bigwedge_{i=1}^{n} \left[\mathit{read}(i) \Rightarrow (x_i = \tilde{x}_i) \right] & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(y_i) \right] & \end{pmatrix} \rightarrow \varphi(\tilde{X}; Y).$$

Testing whether a candidate specification $\psi(X \cup \{r\}; Y) = \mathit{Imp}(\psi_e, \psi_s)$ is an asynchronous strengthening of $\varphi(X;Y)$ is relatively straightforward: The first two conditions in the definition above require identity of propositional formulae, and the third one is LTL formula whose validity could be tested using tools such as TLV and JTLV [PS96, PSZ10].

Given $\psi(X \cup \{r\}; Y) = \mathit{Imp}(\psi_e, \psi_s)$ – an asynchronous strengthening of $\varphi(X;Y)$ – we define a simple kernel formula whose synchronous realizability implies in some cases asynchronous realizability of $\varphi(X;Y)$. This is done by *under-approximating* $\mathcal{X}^{n,m}$. Formally, we have the following.

$$\mathcal{X}_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) = \alpha^{n,m}(r) \rightarrow \beta_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$$

where $\beta_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$ is given by

$$\begin{pmatrix} \mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y}) & \wedge \\ \psi(X \cup \{r\}; Y) & \wedge \\ \bigwedge_{i=1}^{m} \big[[\neg \mathit{write}_n(i) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(y_i)\big] & \end{pmatrix}$$

and where $\mathit{declare}^{n,m}(\{r\}; Y \cup \tilde{Y})$ is given by

$$\begin{pmatrix} \bigwedge_{i=1}^{m} \big[\mathit{write}_n(i) \Rightarrow (y_i = \tilde{y}_i)\big] & \wedge \\ \Big[\Big[(r = \ominus r) \vee \bigvee_{i=1}^{m} (r = (n+i))\Big] \Rightarrow \Big[\bigwedge_{i=1}^{m} (\tilde{y}_i = \ominus \tilde{y}_i)\Big]\Big] & \end{pmatrix}.$$

The formula $\mathit{declare}^{n,m}$ ensures that the declared outputs are updated only at reading points. Indeed, for every i, $\tilde{y}_i$ is allowed to change only when r changes to a value in $\{1, \ldots, n\}$. Furthermore, the outputs themselves copy the value of

the declared outputs (and only when they are allowed to change). It follows that the system 'ignores' inputs that are not at reading points in its next update of outputs.

We note that $\mathcal{X}_{\psi}^{n,m}$ is very similar to $\mathcal{X}^{n,m}$. It replaces the third, 'problematic', conjunct in $\beta^{n,m}$ – the one we named $\beta_3^{n,m}$ earlier – by handling the declared inputs through $declare^{n,m}$.

Since we later show how $\mathcal{X}_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$ is used, in some cases, to conclude that $\varphi(X;Y)$ is asynchronously realizable (which, according to Theorem 3.3 means that $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is synchronously realizable), we refer to $\mathcal{X}_{\psi}^{n,m}$ as a sort of an **under-approximation** (equivalently, strengthening) of $\mathcal{X}^{n,m}$.

In order to use $\mathcal{X}_{\psi}^{n,m}$ as we intend, we need to specify some more characteristics of specifications that would help us in specifying the class of specifications which we have a method of synthesizing. The first definition has to do with specifications that are unaffected by stuttering quantification. In the process of translating synchronous to asynchronous programs (as is the case in this work, where we use techniques to reduce one into the other) we use this characteristic to justify why computations that were 'good' from the perspective of a synchronous program that follows all inputs and all outputs, are also 'good' from the limited, asynchronous, perspective of only values that appear at $I \backslash O$ points, and the other way around. The second case is 'shorter' than the first, so in the process of comparing the two we need to 'stretch' and *squeeze* computations (alternatively, to simply 'stutter'), and this is where the following definition comes handy:

Definition 3.4 (stuttering robustness)**.** *A* LTL *specification* $\xi(X;Y)$ *is* **stutteringly robust** *if for all computations* σ *and* σ' *such that* σ' *is a stuttering variant of* σ, $\sigma, 0 \models \xi$ **iff** $\sigma', 0 \models \xi$.

Stuttering robustness of a specification $\varphi(X;Y)$ could be verified by converting $\varphi(X;Y)$ to a nondeterministic Büchi automaton [VW94], adding to it transitions that capture all stuttering options [PR89b], and then checking that it does not intersect the automaton for $\neg\varphi(X;Y)$. Thus, the complexity of this algorithm is that of LTL satisfiability, it is in PSPACE [VW94]. In our case, when handling LTL specifications with $GR(1)$ winning conditions, in many cases, all the parts of the specifications are relatively simple invariants on the way variables can change and it can be immediately observed that the formula is 'stuttering free'.

Another important characteristic of specifications is memory-lessness, which essentially expresses the notion that it would be possible to make transitions that would not violate a specification, without knowing anything about the history of a computation until that very current state. Such specifications would be easier to work with in a framework where only part of the history (as in asynchronous communications that rely on reading points) are available.

Definition 3.5 (memory-lessness). *A* LTL *specification* φ *is* **memory-less** *if for all computations* $C = c_0, c_1, \ldots$ *and* $C' = c'_0, c'_1, \ldots$ *such that* $C, 0 \models \varphi$ *and* $C', 0 \models \varphi$*, if for some* i *and* j *we have* $c_i = c'_j$*, then the composite computation* $c_0, c_1, \ldots, c_i, c'_{j+1}, c'_{j+2}, \ldots$ *also satisfies* φ*.*

We note that in our case, specifications of the form $\varphi_e = (I_e, S_e, L_e)$ are always memory-less. This is because the syntactic structure of S_e is such that it forces a relation between possible current and primed (next) states that does not further depend on the past, and because, L_e is a conjunction of properties of the form $\Box \Diamond p$, where p is a Boolean formula. Notice that in case the initial specification included past temporal formulae, these are embedded into the variables of the system, and if these variables cannot be viewed by the system then they could

lead to a behavior that is not memory-less. This implies that in case that we want to start from formulae that include past elements and we would like to use a heuristic that relies on memory-lessness we would have to allow the system access to the variables of the temporal testers ([PZ08]) for these past formulae as well.

In the general case, memory-lessness of a specification $\varphi(X;Y)$ can be checked by the following construction. We convert both $\varphi(X;Y)$ and $\neg\varphi(X;Y)$ to nondeterministic Büchi automata N_+ and N_-. Then, we create a nondeterministic Büchi automaton that runs two copies of N_+ and one copy of N_- simultaneously. The two copies of N_+ 'guess' two computations that satisfy $\varphi(X;Y)$ and the copy of N_- checks that the two computations can be combined in a way that does not satisfy $\varphi(X;Y)$. Thus, the language of this product automaton would be empty **iff** $\varphi(X;Y)$ is not memory-less. It follows that memory-lessness can be checked in PSPACE, similarly to LTL satisfiability [VW94].

One important observation about memory-lessness, is that if the specification $\varphi(X;Y) = Imp(\varphi_e, \varphi_s)$ has a memory-less environment φ_e, then also every asynchronous strengthening of $\varphi(X;Y)$ has a memory-less environment. This is since memory-lessness in such a case is only dependent on the safety component of a specification, and this element is identical in any asynchronous strengthening of $\varphi(X;Y)$.

Using this set of definitions, we have the following:

Theorem 3.9. *Let $\varphi(x;Y) = Imp(\varphi_e, \varphi_s)$, where $\varphi_e = \langle I_{\varphi_e}, S_{\varphi_e}, L_{\varphi_e}\rangle$, be a stutteringly robust specification with a $GR(1)$ winning condition and with a memory-less environment, where $|Y| = \{y_1, \ldots, y_m\}$ and where there is exactly one input - x. Let r be a scheduling variable ranging over $\{1, \ldots, (1+m)\}$, and let $\tilde{Y}$ be declared output variables.*

If $\psi(x, r; Y)$ is a stutteringly robust asynchronous strengthening of $\varphi(x; Y)$ such that $\mathcal{X}_{\psi}^{1,m}(x, r; Y \cup \tilde{Y})$ is synchronously realizable, then $\varphi(x; Y)$ is asynchronously realizable.

Furthermore, given a program P_s that synchronously realizes $\mathcal{X}_{\psi}^{1,m}(x, r; Y \cup \tilde{Y})$ it can be converted to a program P_a that asynchronously realizes $\varphi(x; Y)$ in time linear in the number of transitions of the LTS that induces P_s.

The proof of this theorem is constructive, and is given in Subsection 3.6.2 by presenting an algorithm for converting P_s (synchronously realizing $\mathcal{X}_{\psi}^{1,m}$) into P_a (asynchronously realizing φ).

In the following subsection we show how, under some restrictions, we can use Theorem 3.9 together with the algorithm of [PPS06] for $\mathcal{X}_{\psi}^{n,m}$ to test effectively whether its underlying specification φ is asynchronously realizable (given that we found an asynchronous strengthening ψ).

3.6.2 Using the Under-Approximation, and synthesis

By Theorem 3.9, $\mathcal{X}_{\psi}^{n,m}$ could be used to conclude effectively asynchronous realizability of φ, and to build an asynchronously realizing program for it, when φ falls within some restrictions (ψ is an asynchronous strengthening of φ). The most notable of these restrictions is that we consider only single-input specifications that look like this - $\varphi(x; Y)$ (x is a single variable). It is possible that more specifications could be synthesized in a similar manner, but we do not handle such cases.

Since we consider here only single-input specifications, and we assume that any past components of the specifications were eliminated in a pre-processing of the specification, then all of the specifications that we consider have naturally memory-less environments.

We use the algorithm of [PPS06] to analyze synchronous realizability of $\mathcal{X}_{\psi}^{n,m}$ and to synthesize specifications. This algorithm, when it synthesizes a specification, produces a nondeterministic LTS T_s that represents a set of possible programs for $\mathcal{X}_{\psi}^{n,m}$. Since when we synthesize a specification $\varphi(x;Y)$ we are interested in a ILTS that induces a realizing asynchronous program for it, we should start with describing the conversion of T_s into a nondeterministic ILTS T_a that could be determinized into a program for $\varphi(x;Y)$.

For a LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s\rangle$, state $st_{es} \in S_s$ is an *eventual successor* of state $st \in S_s$ if there exists $m \leq |S_s|$ and states $\{s_1, \ldots, s_m\} \subseteq S_s$ such that the following hold: $s_1 = st$ and $s_n = st_{es}$; For all $0 < i < m$, $(s_i; s_{i+1}) \in R_s$; For all $0 < i < m$, if $L(s_1)|_{\{r\}} = r_1$ then $L(s_i)|_{\{r\}} = r_1$, but $L(s_m)|_{\{r\}} \neq r_1$. If $L(s_m)|_{\{r\}} = 1$ we also call st_{es} an *eventual read successor*, otherwise an *eventual write successor*. Note that due to the way the scheduling variable, r, updates in asynchronous interactions, its interpretation in eventual successors of some source state is uniquely defined by its interpretation in that source state.

Given a LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s\rangle$ such that $Y = \{y_1, \ldots, y_m\}$, that was synthesized for the LTL formula $\mathcal{X}_{\psi}^{1,m}(x, r; Y \cup \tilde{Y})$, we define its *implied ILTS* $T_a = \langle S_a, I_a, R_a, \{x\}, Y, L_a, i_a\rangle$, that we shall prove to synthesize $\varphi(x;Y)$. We assume that for (the unique) $i_s \in I_s$, $L_s(i_s)|_Y = Y_{init}$ for some $Y_{init} \in D^Y$[9].

The ILTS *extraction* algorithm in Fig. 3.2 describes the construction of i_a, I_a, L_a, S_a and R_a. In the first part of the algorithm that follows its initialization, between lines 5 and 15, all reading states reachable from I_s are found, and used to

[9] As we already explained in Section 3.3, I_s must be independent of x if $\mathcal{X}_{\psi}^{1,m}$ is synchronously realizable. We further assume that it is a unique assignment due to our simplifying assumption regarding an initial assignments in defining ILTS. The construction of T_a would be very similar if we expand the definition to account for a set of possible initial values, and the proof of asynchronous realizability that follows would still hold. We leave it to readers to expand this algorithm.

build I_a (as part of S_a). In the second part, between lines 16 and 43, the $(m{+}1)$-th eventual successors of each reading state are added to S_a. This second part insures that all writing states are 'skipped' so that R_a transitions include only transitions between consecutive reading states.

In addition to the construction described in Fig. 3.2, we add to S_a D new 'sink' states $sink_d$ (for all $d \in D$). For all $d \in D$, $L_a(sink_d)|_Y$ is defined arbitrarily, and $L_a(sink_d)|_{\{x\}} = d$. For all $d_1, d_2 \in D$, $R_a(sink_{d_1}, sink_{d_2})$. For all $d \in D$, if there exists no $s \in I_a$ such that $L_a(s)|_{\{x\}} = d$, we add to S_a and to I_a a new state i_d, such that $L_a(i_d)|_Y$ is defined arbitrarily, $L_a(i_d)|_{\{x\}} = d$, and for all $d' \in D$, $R_a(i_d, sink_{d'})$. For all $s \in S_a$ and $d \in D$, if there exists no $s' \in S_a$ such that $L_a(s')|_{\{x\}} = d$ and $R_a(s, s')$, then $R_a(s, sink_d)$. These additional states guarantee that T_a is receptive. In the case that T_s was receptive – as it should always be – all of these additions should be already taken care of simply by following the extraction algorithm. We describe them here only for the purpose of expressing that the extracted ILTS handles inputs that violate the environment's initial condition or safety component by continuing to a computation that would remain in sink states.

Constructively, the claim behind Theorem 3.9 is that, in some well-defined cased, the ILTS T_s that is extracted from the LTS T_a produces induced programs that asynchronously realize $\varphi(x;Y)$, as intended. Notice that in the following theorem and proof we name the scheduling variable s, and not r as in the rest of the chapter:

Theorem 3.9 (repeated, with reference to ILTS extraction). *Let* $\varphi(x;Y) = Imp(\varphi_e, \varphi_s)$, *where* $\varphi_e = \langle I_{\varphi_e}, S_{\varphi_e}, L_{\varphi_e} \rangle$, *be a stutteringly robust specification with a* $GR(1)$ *winning condition and with a memory-less environment, where* $|Y| = \{y_1, \ldots, y_m\}$ *and where there is exactly one input -* x. *Let* s *be a scheduling variable*

Input: LTS $T_s = \langle S_s, I_s, R_s, \{x, r\}, Y, L_s \rangle$ such that $|Y| = m$, and an initial outputs assignment Y_{init}.

Output: The elements i_a, I_a, L_a, S_a and R_a of the implied ILTS $T_a = \langle S_a, I_a, R_a, \{x\}, Y, L_a, i_a \rangle$.

1: $i_a \leftarrow Y_{init}$
2: $I_a \leftarrow \emptyset$, $S_a \leftarrow \emptyset$, $R_a \leftarrow \emptyset$
3: $ST \leftarrow$ [`EmptyStack`] ▷ a new states stack (for reachable unexplored 'read' states)
4: $touched \leftarrow \emptyset$ ▷ a new states set (for states that were pushed to ST)
5: **for all** $ini \in I_s$ **do** ▷ find all reachable initial 'read' states
6:   **for all** $succ \in S_s$ s.t. $succ$ is eventual (read) successor of ini **do**
7:     **if** $succ \notin touched$ **then** ▷ add a new state to I_a and S_a
8:       push $succ$ to ST
9:       $touched \leftarrow touched \cup \{succ\}$
10:       $I_a \leftarrow I_a \cup \{succ\}$
11:       $S_a \leftarrow S_a \cup \{succ\}$
12:       $L_a(succ)|_{\{x\}} \leftarrow L_s(succ)|_{\{x\}}$, $L_a(succ)|_Y \leftarrow L_s(succ)|_{\tilde{Y}}$
13:     **end if**
14:   **end for**
15: **end for**
16: **while** $ST \neq$ [`EmptyStack`] **do** ▷ explore all reachable 'read' states
17:   $st \leftarrow$ pop ST
18:   $gen \leftarrow \{st\}$
19:   **for** $i = 1, \ldots, m$ **do** ▷ find all m-th (last 'write') eventual successors of st
20:     $nextgen \leftarrow \emptyset$ ▷ a new states set
21:     **for all** $st_{gen} \in gen$ **do** ▷ find all i-th eventual successors of st
22:       **for all** $succ \in S_s$ s.t. $succ$ is eventual (write) successor of st_{gen} **do**
23:         $nextgen \leftarrow nextgen \cup \{succ\}$
24:       **end for**
25:     **end for**
26:     $gen \leftarrow nextgen$
27:   **end for**
28:   $nextgen \leftarrow \emptyset$ ▷ a new states set
29:   **for all** $st_{gen} \in gen$ **do** ▷ find all 'eventual read successors' of st
30:     **for all** $succ \in S_s$ s.t. $succ$ is eventual (read) successor of st_{gen} **do**
31:       $nextgen \leftarrow nextgen \cup \{succ\}$
32:     **end for**
33:   **end for**
34:   **for all** $st_{ng} \in nextgen$ **do**
35:     **if** $st_{ng} \notin touched$ **then** ▷ add a new state to S_a
36:       push st_{ng} to ST
37:       $touched \leftarrow touched \cup \{st_{ng}\}$
38:       $S_a \leftarrow S_a \cup \{st_{ng}\}$
39:       $L_a(st_{ng})|_{\{x\}} \leftarrow L_s(st_{ng})|_{\{x\}}$, $L_a(st_{ng})|_Y \leftarrow L_s(st_{ng})|_{\tilde{Y}}$
40:     **end if**
41:     $R_a \leftarrow R_a \cup \{(st, st_{ng})\}$ ▷ add a new transition to R_a
42:   **end for**
43: **end while**
44: **return** i_a, I_a, L_a, S_a, R_a

Figure 3.2: Algorithm for extracting T_a from T_s

ranging over $\{1, \ldots, (1+m)\}$*, and let* $\tilde{Y}$ *be declared outputs variables.*

If $\psi(x, s; Y)$ *is a stutteringly robust asynchronous strengthening of* $\varphi(x; Y)$ *such that* $\mathcal{X}_{\psi}^{1,m}(x, s; Y \cup \tilde{Y})$ *is synchronously realizable and where* T_s *is the (nondeterministic) LTS synthesized for it by the algorithm in [PPS06], then the ILTS* T_a*, that is extracted from* T_s*, induces (after determinization) a program that asynchronously realizes* $\varphi(x; Y)$*.*

Proof Sketch: The gist of this proof is that since T_s maintains ψ (on all of its computations), by ψ being an asynchronous strengthening of φ, guarantees that φ would be satisfied by each computation of T_s independently of the input values that are not at reading points. Therefore, each computation of T_s corresponds with some computations of T_a that agree with it on all values at $I \backslash O$ points. This is achieved by the construction of T_a: from each reading point of T_s, it simply 'jumps' to any of the eventual ones, ignoring anything that happens in between, except for producing the outputs as declared by $\tilde{Y}$ at the source state. The declared output variables are exactly used to make sure that all paths in T_s that originate from some reading state would generate identical outputs in the first 'round' of writing points.

The requirement that φ be stutteringly robust in necessary for making sure that not only that values of unobserved inputs do not affect satisfiability of φ, but also the number of such unobserved values between reading points.

The requirement that the environment is memory-less is used when creating the correspondence between a computation of T_s to that of T_a in order to establish the 'correctness' of the latter from the former. Since only partial information is available to the system in the asynchronous setting, memory-lessness is used to justify that any prefix of a T_s computation that reaches some state of T_a would

be just as good for the generation of the future behavior of the environment. In essence, this allows us to 'copy-and-paste' segments of computations of T_s in order to construct one computation of T_a.

Finally, the limitation on φ having a single input comes from the requirement that we can safely 'copy-and-paste' segments of computations of T_s in order to construct one computation of T_a. Since multiple inputs may change between one reading point (of one of them) and another (of another input), and since such changes cannot be monitored asynchronously, our ability to safely select a direction of the computations tree generated by T_s is very limited. It seems very reasonable that further restrictions on the rate, of pattern, of changes allowed to inputs could help extending this technique to multiple variables. It makes sense that every real asynchronous system must make assumptions on the rate of change of each input and limit the ways in which the inputs change together.

Proof: In this proof we refer to the diagram from Fig. 3.3. In this diagram, all states have their variable assignments (labels) written on them, describing the values of the input x, all outputs $y_1, \ldots, y_n$, and the scheduler s. Only on some states we also write the values of the declared outputs $\tilde{Y}$, and to avoid clutter we simply write them separated from the rest of the variables (by a vertical line), as an additional value of outputs (without the '$\sim$' over them).

We work with the nondeterministic LTS (ILTS) T_s (T_a), and show that any computation generated by any program that they could induce satisfies $\varphi(x; Y)$.

A *path* is a segment of a computation. We say that a path π is *safe for the environment*, if $\pi, 0 \models I_{\psi_e} \wedge \Box\, S_{\psi_e}$. (Since ψ is an asynchronous strengthening of φ, this is identical to saying that $\pi, 0 \models I_{\varphi_e} \wedge \Box\, S_{\varphi_e}$.)

Since $\varphi(x; Y)$ has a memory-less environment, and since this implies that

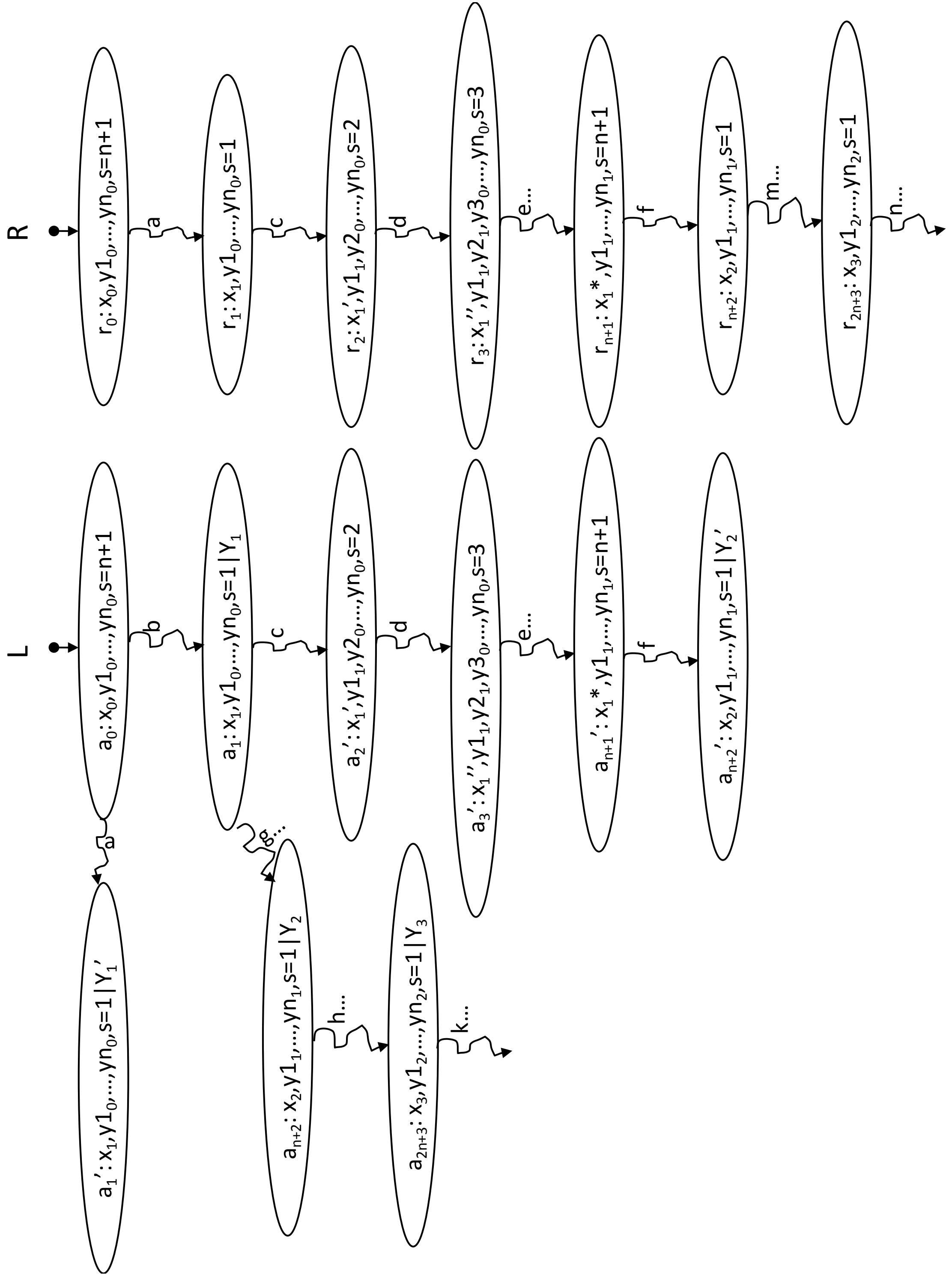


Figure 3.3: The 'real' (R) and T_s (L) computations

$\psi(\{x, s\}; Y)$ also has a memory-less environment, we allow ourselves to simply refer to, at times, a 'memory-less environment', where specifying the relevant specification is not critical.

The computation depicted on the right of Fig. 3.3, named R, with state names starting with the letter r, depicts the 'real' computation in the sense that it shows all of the transitions of both the system and the environment, guided by one program P_a induced by T_a (which considers, naturally, only inputs at reading points). On the left we describe a tree of paths guided by programs induced by T_s, named L, with state names starting with the letter a. L contains another computation that is generated by one program induced by T_s, that is also generated by the corresponding determinized T_a induced program P_a. Through the construction of the computation in L, we show that $R \models \varphi(x; y_1, \ldots, y_n)$.

Since we identify states in R by their assignment to variables, we freely 'borrow' the labeling function L_s to represent their value assignments. We also compare labeling of states from R and from L despite the fact that L labels cover $\tilde{Y}$ while R labels do not, referring only to the shared labels. Finally, we compare labeling of states by T_a and T_s, in the following way: For a state st, $L_a(st) = L_s(st)$ **iff** $L_a(st)|_Y = L_s(st)|_{\tilde{Y}}$ and $L_a(st)|_{\{x\}} = L_s(st)|_{\{x\}}$.

Let r^0, with $L_s(r^0) = \langle x^0, s^0, y_1^0, \ldots, y_n^0 \rangle$, also written as $L_s(r^0) = \langle x^0, s^0, Y^0 \rangle$ ($s^0 = n{+}1$), be the first state in R. If $L_s(r^0) \not\models I_e$, then $R, 0 \models \varphi(x; Y)$. Otherwise, due to the receptiveness of T_s, the state r^0 is also some initial state $a^0 \in I_s$ ($L_s(a^0) = L_s(r^0)$), and, by construction, $L_s(a^0) \models I_{\psi_e} \wedge I_{\psi_s}$. Also by construction, $i_a = Y^0$ (since we assume a single possible initial output, if $L_s(a^0) \not\models I_e$ then this must be the case), and T_a outputs a 'good' initial value. All paths in R and in L satisfy, therefore, $I_{\psi_e} \wedge I_{\psi_s}$.

If any prefix of R is not safe for the environment then, trivially, $R, 0 \models \varphi(x; Y)$. In the following, we assume, for completeness, that all prefixes of R are safe for the environment.

The path of R up until the first reading point, $r^0 \rightsquigarrow a$, exists in T_s - $a^0 \rightsquigarrow a$ (since T_s, by construction, induces programs that provide outputs against any behavior of the environment). Similarly, a state that agrees with the next state of R, r^1, on its labeling of $\{x, s\} \cup Y$ must be reachable (in one transition) from the last state of a in T_s - $a^0 \rightsquigarrow a \rightsquigarrow a^{1'}$ ($L_s(a^{1'})$ has some labeling for $\tilde{Y}$ - $\tilde{Y}^{1'} = \langle y_1^{1'}, \ldots, y_n^{1'} \rangle$). It is important to note here that both T_a and T_s might be non-deterministic. Let a^1 be some eventual successor of a^0 in T_s (through the path $a^0 \rightsquigarrow b \rightsquigarrow a^1$), that is also in T_a and that agrees with r^1 (and with $a^{1'}$) on the labeling of $\{x, s\} \cup Y$. By construction, both $a^{1'}$ and a^1 are in I_a, and T_a has 'good' initial states. We assume that, in this computation, a^1 was 'chosen'. Say that a^1 has the labeling of $\tilde{Y}$ $\tilde{Y}^1 = \langle y_1^1, \ldots, y_n^1 \rangle$. Let c be the path of R from r^1 up until the first writing point. Since the environment is memory-less (second assumption), and since r^1 and a^1 agree on the labeling of all the variable of S_{ψ_e}, the transition from a^1 to the first state of c must exist in T_s, and so are the rest of the transitions of c.

Let r^2 be the first writing state in R that immediately follows the last state of c, where the label of y_1 is changed from y_1^0 to y_1^1 (according to $\tilde{Y}^1$, as controlled by T_a through the selection of the eventual successor a^1 of a^0). Since (by assumption) the transition in R from the last state of c to r^2 is safe for the environment (note here that S_{ψ_e} is independent of Y' - the primed version of Y), than a transition from the last state of c in the path $a^0 \rightsquigarrow b \rightsquigarrow a^1 \rightsquigarrow c$ in T_s, to a state that agrees with r^2 on its labeling of x, must exist. Moreover, all eventual successors of a^1 in

T_s must label the value y_1^1 to y_1. Let $a^{2'}$ be one such eventual successor of a^1 in T_s, that agrees with r^2 on its labeling of x (by the construction of T_a, that replaces its outputs labeling with the declared outputs labeling of T_s, it must exist in a path that is driven by T_a).

As R continues with a path that, starting from r^1, writes all outputs in writing points ($r^1 \rightsquigarrow c \rightsquigarrow r^2 \rightsquigarrow d \rightsquigarrow r^3 \rightsquigarrow e \ldots \rightsquigarrow r^{n+1}$), for similar arguments of memory-lessness of the environment and of agreement with the states of R on their labeling of $\{x, s\} \cup Y$, a similar path must exist in L - $a^1 \rightsquigarrow c \rightsquigarrow a^{2'} \rightsquigarrow d \rightsquigarrow a^{3'} \rightsquigarrow e \ldots \rightsquigarrow a^{(n+1)'}$. Moreover, the following path of R that originates in r^{n+1} and that continues up until, and including, the next reading point r^{n+2} - $r^{n+1} \rightsquigarrow r^{n+2}$, must also be duplicated in T_s - $a^{(n+1)'} \rightsquigarrow a^{(n+2)'}$ ($a^{(n+2)'}$ has some value $\tilde{Y}^{(n+2)'} = \langle y_1^{(n+2)'}, \ldots, y_n^{(n+2)'} \rangle$). All states on the path from r^1 to r^{n+1} agree with all states on the path from a^1 to $a^{(n+2)'}$, respectively, on their labels of $\{x, s\} \cup Y$ (including the 'writing' of the same output values at all writing points, and 'reading' of the same input value at the terminating reading point).

Since the state $a^{(n+2)'}$ exists in T_s then, by construction of T_a, at least one sequence of $n + 1$ consecutive eventual successors from T_s that originate from a^1 (which is in T_a) would be represented in T_a through the value of $L_a(a^1)|_Y$, 'writing' in all writing points along the way all the values of $\tilde{Y}^1$ (one-by-one), and terminating in the state $a^{(n+2)}$ that agrees with $r^{(n+2)}$ on its labeling of $\{x, s\} \cup Y$. Say that $L_s(a^{(n+2)})|_{\tilde{Y}}$ is the labeling $\tilde{Y}^{(n+2)} = \langle y_1^{(n+2)}, \ldots, y_n^{(n+2)} \rangle$.

We showed that a path that corresponds to the writing-then-reading path $c \rightsquigarrow r^2 \rightsquigarrow d \rightsquigarrow r^3 \rightsquigarrow e \ldots \rightsquigarrow r^{n+1} \rightsquigarrow f \rightsquigarrow r^{(n+2)}$ of R exists in T_s and, therefore, represented in T_a. We continue inductively to construct a computation in T_s, called S_c, that is driven by one program induced by t_a - $a^0 \rightsquigarrow b \rightsquigarrow a^1 \rightsquigarrow g \ldots \rightsquigarrow$

$a^{(n+2)} \rightsquigarrow h \ldots \rightsquigarrow a^{(2n+3)} \rightsquigarrow k \ldots.$

The key observation about S_c, other than the fact that it must exist (assuming that R is safe for the environment), is that it agrees with R on its interpretations of Y at all corresponding states, and that it agrees with R on its interpretations of x at all corresponding reading states. We know that the following holds for S_c (using L_s, and as for all T_s computations)

$$S_c, 0 \models \begin{pmatrix} \alpha^{1,m}(s) & \wedge \\ I_{\psi_e} \wedge \Box\, S_{\psi_e} & \wedge \\ \psi(X \cup \{s\}; Y) & \wedge \\ \bigwedge_{i=1}^{m} \left[[\neg \mathit{write}_1(i) \wedge \neg \mathit{first}] \Rightarrow \mathit{unchanged}(y_i)\right] & \end{pmatrix}$$

Since ψ is an asynchronous strengthening of φ (specifically, due to the implication that is in that definition), we conclude that $S_c, 0 \models [\mathit{read}(1) \Rightarrow (x = \tilde{x})] \to \varphi(\tilde{x}; Y)$. Using the observation regarding the connection between R and S_c, we conclude that $R, 0 \models [\mathit{read}(1) \Rightarrow (x = \tilde{x})] \to \varphi(\tilde{x}; Y)$ and, in particular, that $R, 0 \models \varphi(x; Y)$. (Actually, in the transition from S_c to R we must account for the fact that the two computations may differ in length between every $I \backslash O$ point. To overcome that we use the stuttering robustness of φ, ψ and the rest of the clauses from the definitions of asynchronous strengthening.) $\square$

If $\varphi(x; Y)$ has a $GR(1)$ winning condition and if $\psi(x, r; Y)$ is an asynchronous strengthening of $\varphi(x; Y)$, then $\mathcal{X}_{\psi}^{1,m}(x, r; Y \cup \tilde{Y})$ has a $GR(1)$ winning condition. The work in [PPS06] provides a $O(N^3 \cdot m \cdot n)$-time method for verifying that a specification with a $GR(1)$ winning condition is synchronously realizable, and for synthesizing a synchronously realizing program. (As before, N is the state space

of the specification, and m and n are the number of liveness conjuncts of the environment and system's specifications, respectively.) Thus, we can effectively check whether the kernel formula $\mathcal{X}_{\psi}^{1,m}$ is synchronously realizable. In the case that it is, and if $\varphi(x;Y)$ is stutteringly robust with a memory-less environment, we can conclude, using Theorem 3.9, that $\varphi(x;Y)$ is asynchronously realizable and we can effectively construct an asynchronously realizing program for it.

Note that while the proposed method is sound, it is not complete, in the sense that a specification $\varphi(x;Y)$ may be asynchronously realizable but the derived synchronous approximation for its asynchronous strengthening $\psi(x,r;Y)$, namely, $\mathcal{X}_{\psi}^{1,m}$ may be synchronously unrealizable. One possible solution to such cases might be searching for another asynchronous strengthening.

Caveat: There exist specifications which the effective algorithm we propose – as described in [PPS06] – declare synchronously unrealizable, while they are in fact realizable. This may lead to a false classification of $\mathcal{X}_{\psi}^{1,m}$ as synchronously unrealizable, and therefore to an inability to recognize that the underlying specification φ might be asynchronously realizable. Such specifications with $GR(1)$ winning conditions are characterized in Chapter 2, where an efficient method is described for identifying them, as well as a method for avoiding false classifications given such specifications.

In the context of synthesizing an asynchronously realizing program for a specification, we define the set of *interface variables* to be the set of variables of its relevant asynchronous strengthening. These are the same variables of the original specification, with the addition of the scheduling variable.

One important observation regarding the LTS T_s that was synthesized for $\mathcal{X}_{\psi}^{1,m}(x,r;Y \cup \tilde{Y})$, is that it contains labels for the declared outputs. These la-

bels are redundant since they could always be picked up from the outputs of any m consecutive eventual write successors of any reading point (in any non-writing point they remain unchanged). Therefore, they may be dropped from T_s. As mentioned in Subsection 3.2.3, in practice often specifications are written using past formulae and we must incorporate into them temporal testers [PZ08] in order to bring them to the *Imp* formula structure. In such cases, the interface variables of the original specification would be a subset of the interface variables in the 'translated' structure. However, as long as the formula that is synthesized falls within the restrictions that we define for successful synthesis, one should have no problem expressing the resulting asynchronous program without any variables that were added for temporal testers. Once T_s is constructed, therefore, labels for such variables may be dropped as well.

Instead of simply dropping the not-needed labels from T_s, one beneficial process would be to minimize T_s to represent only the perspective of the interface variables. Minimizing T_s before the construction of T_a results in an overall shorter construction time and, more importantly, is a smaller (and, therefore, more efficient) ILTS T_a that corresponds to a program for asynchronously realizing $\varphi(x;Y)$. One way to perform such a minimization is by using a variant of the Myhill-Nerode minimization for deterministic finite-state automata.

3.6.3 Applying the Realizability Test

In this subsection we illustrate the application of the effective realizability test, and synthesis, based on Theorem 3.9.

If one wishes to use the method for verifying asynchronous realizability and for synthesis that is proposed in this section, the first question to be answered is –

how can one come up with an asynchronous strengthening? One possible heuristic that may be applied for this purpose is given by:

Heuristic 3.1. *In order to derive an asynchronous strengthening* $\psi(X \cup \{r\}; Y)$ *for a specification* $\varphi(X; Y)$ *(where* $|X| = n$ *and* $|Y| = m$ *and with a scheduling variable* r*), replace one or more occurrences of atomic formulae of inputs, e.g.,* $x_i = d$*, by* $(x_i = d) \wedge \ominus(r \neq i) \wedge (r = i)$*, which means that* $x_i = d$ *at a reading point.*

The rationale for this heuristic is that since ψ is supposed to imply both $\varphi(X; Y)$ and $\varphi(\tilde{X}; Y)$, and the only information we have about $\tilde{X}$ is that it agrees with X at all reading points, it is natural to transform references to X to references to X at a reading point.

We start with the 'response' specification $\varphi_3(x; y) = \mathit{Imp}(\varphi_{3,e}, \varphi_{3,s})$. This specification has a $GR(1)$ winning condition, it is stutteringly robust with a memoryless environment, and therefore it is potentially a good candidate to use with Theorem 3.9. The first step is to identify an asynchronous strengthening of it $\psi_3(x, r; y) = \mathit{Imp}(\psi_{3,e}, \psi_{3,s})$. Applying Heuristic 3.1 to φ_3, we obtain the proposed specification $\psi_3(x, r; y)$ given by

$$[\neg(x \leftrightarrow y) \Rightarrow (x \leftrightarrow \bigcirc x)] \rightarrow \left(\begin{array}{ll} x \Rightarrow \Diamond y & \wedge \\ \overline{x} \Rightarrow \Diamond \overline{y} & \wedge \\ y \Rightarrow y \, \mathcal{S} \, \overline{y} \, \mathcal{S} \, [x \wedge \ominus(r = 2) \wedge (r = 1)] & \wedge \\ \bigcirc\{\overline{y} \Rightarrow \overline{y} \, \mathcal{B} \, y \, \mathcal{S} \, [\overline{x} \wedge \ominus(r = 2) \wedge (r = 1)]\} & \end{array}\right)$$

Note that the occurrences of references to x that have been replaced are the references to x within the sub-formulae $(y \Rightarrow y \, \mathcal{S} \, \overline{y} \, \mathcal{S} \, x)$ and $\bigcirc(\overline{y} \Rightarrow \overline{y} \, \mathcal{B} \, y \, \mathcal{S} \, \overline{x})$.

We observe that $I_{\psi_{3,e}} = I_{\varphi_{3,e}}$ and that $S_{\psi_{3,e}} = S_{\varphi_{3,e}}$. To establish that ψ_3 is an

asynchronous strengthening of φ_3 it is left to check that the specifications satisfy the implication that is in the definition for asynchronous strengthening. This has been done by using the tool TLV [PS96].

Applying the synchronous realizability test of [PPS06] to the kernel formula $\mathcal{X}_{\psi_3}(x, r; y)$ (derived for ψ_3), the algorithm informs us that this specification is synchronously realizable and produces a realizing LTS S_3 with 30 states and 90 transitions (not including the *sink* state). Applying a minimization procedure to this LTS yields a minimal LTS S_3' with 16 states and 54 transitions. Applying the extraction algorithm that builds the ILTS $A_{S_3'}$ from S_3', yielded a nondeterministic ILTS with all 16 states and 54 transitions. A simplified sub-ILTS of $A_{S_3'}$ that provides a complete strategy for $\varphi_3(x; y)$ is presented in Fig. 3.4 as an automaton.

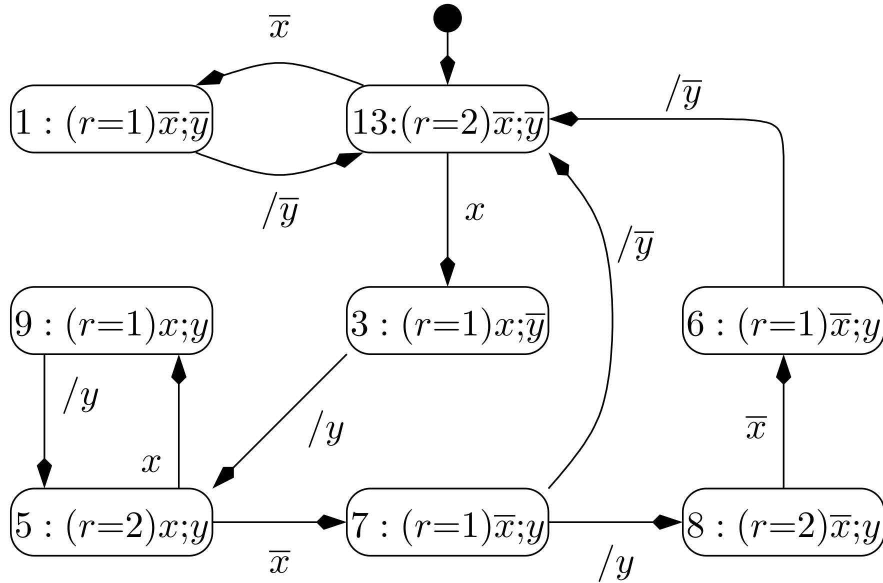


Figure 3.4: ILTS (as an automaton)

Note that the automaton of Fig. 3.4 has a certain degree of nondeterminism as

demonstrated in the exits out of state 7. At this point, it may nondeterministically choose to output y or $\overline{y}$. The automaton of Fig. 3.4 is a simplified version of the sub-automaton of $A_{S'_3}$. In particular, we have identified some states that have been found to have equivalent behavior. This led to the fact that the automaton does not contain the initial state 0 that has been identified with state 13 which is taken to be the initial state. Also, to avoid clutter, we omitted the representation of the *sink* state and all transitions entering it.

We model-checked [CGP99] and made sure that all possible computations driven by $A_{S'_3}$, and within the simplified automaton from Fig. 3.4, satisfy $\varphi_3(x; y)$ (again, using TLV [PS96]).

To make sure that our method works also for multiple outputs, we constructed two new specifications, $\varphi_4(x; y_0, y_1)$ and $\varphi_5(x; y_0, y_1, y_2)$, that expand the logic of the 'response' specification $\varphi_3(x; y)$ to two, and three, outputs, respectively. Unlike in $\varphi_3(x; y)$, the inputs of these two new specifications are not Boolean (their outputs, however, are). Both specifications have $GR(1)$ winning conditions, are stutteringly robust and have a memory-less environment. Therefore they are good candidates to use with Theorem 3.9.

The input x in $\varphi_4(x; y_0, y_1)$ ranges over $\{0, 1, 2, 3\}$. It's intended meaning is that $x = 0$ is interpreted as a 'request' to output $\overline{y_1}$, $x = 1$ to output y_1, $x = 2$ to output $\overline{y_0}$, and $x = 3$ to output y_0. $\varphi_4(x; y_0, y_1) = \mathit{Imp}(\varphi_{4,e}, \varphi_{4,s})$, where

$$\varphi_{4,e}(x; y_0, y_1) = \begin{pmatrix} ((x = 0) \wedge y_1) & \vee \\ ((x = 1) \wedge \overline{y_1}) & \vee \\ ((x = 2) \wedge y_0) & \vee \\ ((x = 3) \wedge \overline{y_0}) & \end{pmatrix} \Rightarrow \bigcirc\, \mathit{unchanged}(x)$$

and where

$$\varphi_{4,s}(x; y_0, y_1) = \begin{pmatrix} (x = 0) \Rightarrow \Diamond\, \overline{y_1} & \wedge \\ (x = 1) \Rightarrow \Diamond\, y_1 & \wedge \\ (x = 2) \Rightarrow \Diamond\, \overline{y_0} & \wedge \\ (x = 3) \Rightarrow \Diamond\, y_0 & \wedge \\ y_0 \Rightarrow y_0\, \mathcal{S}\, \overline{y_0}\, \mathcal{S}\, (x = 3) & \wedge \\ y_1 \Rightarrow y_1\, \mathcal{S}\, \overline{y_1}\, \mathcal{S}\, (x = 1) & \wedge \\ \bigcirc[\overline{y_0} \Rightarrow \overline{y_0}\, \mathcal{B}\, y_0\, \mathcal{S}\, (x = 2)] & \wedge \\ \bigcirc[\overline{y_1} \Rightarrow \overline{y_1}\, \mathcal{B}\, y_1\, \mathcal{S}\, (x = 0)] & \end{pmatrix}$$

(For clarity, we do not specify here an initial condition - $(x = 0) \wedge \overline{y_1} \wedge \overline{y_2}$.)

An asynchronous strengthening $\psi_4(x, r; y_0, y_1)$ is constructed for φ_4 using Heuristic 3.1 in a similar manner to the construction of ψ_3. Applying the synchronous realizability test of [PPS06] to the kernel formula $\mathcal{X}_{\psi_4}(x, r; y_0, y_1)$ (derived for ψ_4), the algorithm informs us that this specification is synchronously realizable and produces a realizing LTS S_4 with 340 states and 1544 transitions (not including a *sink* state). Applying a minimization procedure to this LTS yields a minimal LTS S'_4 with 196 states and 1056 transitions. Applying the extraction algorithm that builds the ILTS $A_{S'_4}$ from S'_4, yielded a nondeterministic ILTS that model-checked positively to satisfy φ_4 along all of its computations (generated through asynchronous interactions).

The specification $\varphi_5(x; y_0, y_1, y_2)$, that follows the exact same structure of φ_4 with one additional output (and with an input x that ranges over a domain of size 6), was tested in the same manner and proved to be asynchronously realizable. The LTS S_5 that was constructed for its asynchronous strengthening $\psi_4(x, r; y_0, y_1, y_2)$ (more accurately, for the relevant kernel formula $\mathcal{X}_{\psi_4}$), has 1984 states and 11768

transitions, and after minimization S'_5 has 1184 states and 8680 transitions. The ILTS $A_{S'_5}$ from S'_5 was proven, as before, to implement an asynchronously realizing program for φ_5.

Clearly, the most obvious drawback of this synthesis method is the disproportionately large asynchronous programs (os ILTS) that it creates. However, in this work we did not make any attempt at constructing efficient, or minimal, realizing programs. Any approach to extracting deterministic sub-ILTS from these large nondeterministic ones could help with size reduction, and beyond that would be an issue for future work.

3.6.4 A Possible Direction for Handling More Specifications

The following theorem may provide some hint to expand our under-approximation approach to handle more specifications:

Theorem 3.10. *Let $\varphi(X;Y)$ be a specification with a $GR(1)$ winning condition, where $|X| = n$ and $|Y| = m$, let r be a scheduling variable ranging over $\{1, \ldots, (n+m)\}$, and let $\tilde{Y}$ be a set of declared output variables. If $\psi(X \cup \{r\}; Y)$ is a stutteringly robust asynchronous strengthening of $\varphi(X;Y)$, and if $\Box S_{\psi_e}$ is also stutteringly robust, then the following implication is valid:*

$$\begin{pmatrix} \mathcal{X}_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y}) & \wedge \\ I_{\psi_e} \wedge \Box S_{\psi_e} & \end{pmatrix} \rightarrow \mathcal{X}^{n,m}(X \cup \{r\}; Y)$$

($\mathcal{X}_{\psi}^{n,m}(X \cup \{r\}; Y \cup \tilde{Y})$ and $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$ are the appropriate kernel formulae derived for $\psi(X \cup \{r\}; Y)$ and for $\varphi(X;Y)$, respectively.)

Proof: Let σ be some computation over the variables $X \cup \{r\}; Y \cup \tilde{Y}$. If $\sigma, 0 \not\models \alpha^{n,m}(r)$, then the implication is trivially valid. Otherwise, the left-hand-side

of the implication guarantees all of the following (using the implication structure of $\mathcal{X}_{\psi}^{n,m}$): $\sigma, 0 \models I_{\psi_e}$, $\sigma, 0 \models \Box S_{\psi_e}$, $\sigma, 0 \models \psi(X \cup \{r\}; Y)$, and $\sigma, 0 \models \bigwedge_{i=1}^{m}[(\neg write_n(i) \wedge \neg first) \Rightarrow unchanged(y_i)]$.

To reach the conclusion that $\sigma, 0 \models \mathcal{X}^{n,m}(X \cup \{r\}; Y)$ is valid, the only clause of $\mathcal{X}^{n,m}$ that is left to be proven to be satisfied by σ is $\beta_3^{n,m}$ ($\varphi(X; Y)$ is implied by $\beta_3^{n,m}$).

Let σ' be a stuttering variant of σ. We know that ψ and $\Box S_{\psi_e}$ are stutteringly robust, by assumption, and, therefore, they are satisfied by σ'. Since writing points are transitions in which the scheduling variable r changes its value, all writing points of σ are kept, in order, in σ' (since the *squeeze* operation never merges consecutive states that differ in any variable's value, and since σ' may contain only variants of $squeeze(\sigma)$ that have repetitions of states without additional value changes, specifically not adding new $I \backslash O$-points). A similar statement is true for points where $unchanged(y_i)$ holds, and we get that $\sigma', 0 \models \bigwedge_{i=1}^{m}[(\neg write_n(i) \wedge \neg first) \Rightarrow unchanged(y_i)]$. $\alpha^{n,m}$ is stutteringly robust as well, for a similar argument, and so is I_{ψ_e}, as a non-temporal formula.

We know that the implication that is in the definition for asynchronous strengthening is valid. Particularly, it is satisfied by σ', and by any $\tilde{X}$-variant of it σ''. Since ψ, $\Box S_{\psi_e}$, $\bigwedge_{i=1}^{m}[(\neg write_n(i) \wedge \neg first) \Rightarrow unchanged(y_i)]$, $\alpha^{n,m}$, and I_{ψ_e} are all satisfied by σ', and since non of them contains any variables from $\tilde{X}$, they are all satisfied also by σ''.

So if σ'' satisfies the entire implication from the definition for asynchronous strengthening, and if it also satisfies all but one of the conjuncts on the left-hand-side of that implication, then we can 'remove' them all and get that $\sigma'', 0 \models \bigwedge_{i=1}^{n}[read(i) \Rightarrow (x_i = \tilde{x}_i)]$.

This means, however, that $\sigma, 0 \models (\forall^{\approx} \tilde{X}). \bigwedge_{i=1}^{n} [read(i) \Rightarrow (x_i = \tilde{x_i})]$, which is exactly $\sigma, 0 \models \beta_3^{n,m}$ as we wanted.

□

We provide this theorem here only since it seems reasonable that further analysis of various specifications classes may leverage it to allow for synchronous realizability testing of more specifications.

A careful reader might observe that had we removed from the definition of asynchronous strengthening the requirement the φ and ψ have $GR(1)$ winning conditions, or that they are at all given in the form of *Imp*, we would be able to prove a stronger version of Theorem 3.10, where synchronous realizability of $\mathcal{X}_{\psi}^{n,m}$ implies that of $\mathcal{X}^{n,m}(X \cup \{r\}; Y)$. However, removing $\alpha^{n,m}$, I_{ψ_e}, and S_{ψ_e} from the definition of asynchronous strengthening, and, therefore, from the application that is included in that definition, would make it significantly more difficult to identify 'good' asynchronous strengthening formulae.

3.7 Conclusions and Future Work

In this chapter we considered methods for the 'effective' synthesis of asynchronous systems (programs) from their temporal property specifications. The methods are based on an expansion to the Rosner reduction which converts one specification φ into another specification $\mathcal{X}$, such that $\mathcal{X}$ is synchronously realizable **iff** φ is asynchronously realizable. Roughly speaking, this reduction is based on the implicit simulation of all possible asynchronous behaviors of φ within a synchronous behavior of $\mathcal{X}$, through the addition of a scheduling variable that models all possible context switches from the perspective of the synthesized system.

While the Rosner reduction handled specifications with a single input and a single output, we presented an expansion that handles multiple-variables' specifications – $\mathcal{X}^{n,m}$. We further presented an alternative reduction – $\mathcal{Y}^{n,m}$ – and proved that despite the fact that it appears to be less restrictive and more intuitive, it is, in fact, just as expressive (in terms of realizability) as the first expanded Rosner reduction ($\mathcal{X}^{n,m}$) that provides a canonical, easy to follow, way of describing asynchronous interactions of a system with its environment.

This expanded reduction cannot be applied directly in an efficient way, because the translation of φ into $\mathcal{X}^{n,m}$ involves a special kind of existential quantification. Handling this quantification using the usual automata theoretic approach from [PR89a] leads to very complex algorithms even when the initial formula is relatively simple. Consequently, we proposed a family of formulae of moderate sizes that bound $\mathcal{X}^{n,m}$ from two sides:

- In order to establish unrealizability, it is sufficient to show that the over-approximation $\mathcal{X}^{n,m}_{\downarrow}$ is unrealizable.
- In order to perform synthesis (and establish realizability) for specifications with a single input variable (under some additional contraints), it is sufficient to show that the under-approximation $\mathcal{X}^{1,m}_{\psi}$ is realizable, after identifying a 'strengthening' of φ that is called ψ.

In both cases, if φ belongs to the restricted class of specifications with GR(1) winning conditions, the complexity of these two processes is polynomial and comparable to the tractable complexity of the synchronous synthesis case.

We believe that there is still much room to explore cases in which asynchronous synthesis can be approximated by heuristics that perform well in practice. In

particular, the combination of the two requirements, that the environment has one variable, and that this variable behaves memory-lessly, is very severe. We are still trying to remove these restrictions or parts thereof. We believe that in many cases the synchronous controller produced using our technique still contains the right options for the asynchronous controller. However, we have been unable to identify a constructive way to find these choices. Currently, any eventual read successor in the LTS is a good one in the ILTS. It seems reasonable that, in some cases, which are not memory-less, although an arbitrary choice is not sufficient there may be a way to choose successors wisely (with some lookahead method) and be able to synthesize an ILTS that would not get stuck for every environment behavior.

Chapter 4

Formalization and Automated Verification of RESTful Behavior

4.1 Introduction

REST – an acronym for REpresentational State Transfer – is a software architectural style that is used for the creation of highly scalable web applications. It was formulated by Roy Fielding in [Fie00]. The REST style provides a uniform mechanism for access to resources, thereby simplifying the development of web applications. Its structure ensures effective use of the Internet, in particular of intermediaries such as caches and proxies, resulting in fast access to applications. Over the past decade, interest in REST has increased rapidly, and it has become the desired standard for the development of large-scale web applications. The flip side to this is a considerable confusion over the principles of RESTful design, which are often misunderstood and mis-applied. This results in applications that are functionally correct, but which do not achieve the full benefits of flexibility and scalability that are possible with REST. Fielding has criticized the design of sev-

eral applications which claim to be RESTful, among those are the photo-sharing application Flickr [Fie08a] and the social networking API SocialSite [Fie08b].

The criticisms show that some of the confusion is between REST and the Hypertext Transfer Protocol (HTTP) [FGM+99]. (Aside: Fielding is also a co-author of the HTTP RFC.) While RESTful applications are implemented using HTTP, not every HTTP-based application is RESTful, and not every RESTful application must use HTTP: REST is an architectural style, while HTTP is a networking protocol. Another common mistake is to call a application RESTful if it uses simpler encodings than those in the Remote Procedure Call (RPC) based SOAP/WSDL [W3C07] mechanism. The distinction goes far beyond this superficial difference. These and other, more subtle, confusions motivate our work.

A question which arises naturally is whether it is possible to automatically check an application for conformance to REST. Doing so requires a precise specification of REST. In this chapter, we address both questions. A formal characterization of REST has benefit beyond its use in automated analysis. It should also result in clear and effective communication about REST, and can enable deeper analysis of this elegant and effective architectural style.

We begin by formulating RESTful behavior in a general setting. A key contribution is to show that REST can be formalized within temporal logic. Two constraints define RESTful behavior. One, statelessness, is a branching-time property. The other, hypertext-driven behavior, is expressible in linear temporal logic. Both are safety properties. We then consider the common case of RESTful HTTP, and discuss how HTTP induces variants of the temporal properties.

The temporal specifications may be applied in several ways for verifying that a client-server application is RESTful. One is to model-check a fixed instance of

the application [CE81, QS82]. The parameterized model checking question is also of much interest, as web applications typically handle a large number of clients. These questions presume a 'white-box' situation, where implementation code is available for analysis. A second group of questions concern run-time checking of RESTful behavior, a 'black-box' approach, where the only observable is the client-server communication. A third group of questions concern the synthesis of servers which meet a specification under RESTful constraints.

We show that, for a fixed instance, model-checking statelessness can be done in time that is linear in the size of the state-space of the instance and polynomial in the number of resources. On the other hand, checking that an instance satisfies a specification assuming hypertext-driven client behavior is PSPACE-complete in the number of resources. This property can be checked at run-time, however, in time that is polynomial in the number of clients and resources. We show decidability for parameterized model-checking under certain assumptions; the general case remains open.

This chapter is based on published work; Parts of what presented here were published as a conference paper [KN11a], as well as in a technical report [KN11b].

4.2 REST and its Formalization

Our goal in the formalization is to stay as close as is possible to its description by Fielding in [Fie00], which should be consulted for the rationale behind REST.

4.2.1 Building Blocks for REST

REST is built around a client-server model which includes intermediate components, such as proxies and caches. An application is structured as a (conceptually)

single *server component* (server, for short) and a number of *client components* (clients). All relevant communication is between a client and the server. Each *request* for a service is sent by a client to the server, which may either reject the request or perform it, returning a *response* in either case to the client. A server manages access to *resources*. A resource is an abstract unit of information with an intended meaning. Examples are a data file, a temporal service (e.g., 'current time in France'), or a collection of other resources (e.g., 'all files in a directory').

An *entity* describes the value of a resource at a given time; it can be viewed as the *state* of a resource. A resource state may be constant (e.g., 'Uri's birth date') or changing (e.g., 'current time in France'), but it must take on values which correspond to the intended meaning of the resource. A state may contain both uninterpreted data and links to other resources. This creates a 'Linked Data' view [BHIBL08] of all the information under the control of an application. A *resource identifier* (resource id, for short) is a name by which a resource is identified. The mapping of names to resources is fixed and unique. In HTTP-based applications, Uniform Resource Identifiers (URIs) [W3C05] are the resource identifiers. A *resource representation* is a description of the state of the resource at a given time. A state may have multiple representations (e.g., a web page may be represented as HTML, or by an image of its content).

A RESTful architecture has a fixed set of uniform *methods*. Hence, every application following that architecture must be based on these methods, which effectively decouples interface from implementation. In contrast, for an abstract data type or RPC model, the method set is unconstrained. Properties of a method, such as *safety* (no invocation changes server state) and *idempotence* (repeated invocation does not change server state) are required to hold uniformly, i.e., for all

instantiations of the method.

4.2.2 Formalizing Resource-Based Applications

A resource-based application is one that is organized in terms of the previously described building blocks, which are formally defined by a *resource structure*: a tuple $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, where R is a set of resources; I is a set of resource identifiers; $B \subseteq I$, is a **finite** set of *root identifiers*; $\eta : I \mapsto R$ is a *naming function*, mapping identifiers to resources, a partial function that is injective on its domain; C is a set of *client identifiers*; D is a set of *data values*, with an *equivalence* relation $\sim \subseteq (D \times D)$; OPS is a finite set of methods; and $RETS$ is a finite set of *return codes*.

For simplicity, we use a specific form of resource representation, a pair $\langle ids; d \rangle$ in $2^I \times D$. Here, ids is a set of resource identifiers, and d a piece of data. This abstracts from HTML or XML syntax and formatting, and clearly separates resource identifiers from data values. The relation '$\sim$' may be used to ignore irrelevant portions of data, such as counters or timestamps. We extend it to resource representations as $\langle ids_1; d_1 \rangle \sim \langle ids_2; d_2 \rangle$ **iff** $ids_1 = ids_2$ and $d_1 \sim d_2$.

A *client-server communication* (a communication, for short) is represented by a 'request/response' pair, with the syntax c::$op(i, args)/rc(rvals)$, where: $c \in C$ is a client identifier; $op \in OPS$ is a method; $i \in I$, is a *target resource identifier*; $args$ is a finite list of *arguments*; $rc \in RETS$ is a return code; and $rvals$ is a finite list of *return values*. The arguments and return values are specific to the method. Both may include resource identifiers, data values, and resource representations. (We omit more complex data types for simplicity.)

With each communication m are associated two *disjoint* sets of resource iden-

tifiers, denoted $L(m)$ (linked) and $UL(m)$ (unlinked). The set $L(m)$ describes resources that are made known to the requesting client, and includes resource identifiers which are returned as results in the communication, or that are created by it. The set $UL(m)$ are identifiers which are revoked at the client.

Given a resource structure RS, a *RS-family* is a collection of client and server processes, defined over elements of RS. A *RS-instance* is a specific choice of clients and a single server from a RS-family, with the processes interacting using CCS-style synchronization [Mil89] on communications. A *global state* of a RS-instance is given by a tuple with a local state for the server process and a local state for each client process. A *computation* is an alternating sequence of global states and *actions*, where an action is either a (synchronized) communication between a client and the server, or an internal process transition.

Caveats: In reality, requests and responses are independent events, which allows the processing of concurrent requests to overlap in time. The issue is discussed further in Section 4.4, as treating it directly considerably complicates the model. There is also an implicit assumption that methods have immediate effects. In practice, (e.g., HTTP DELETE) a server may return a response but postpone the effect of a request. This issue is discussed in Subsection 4.3.3.

A *communication sequence* σ is a (possibly infinite) sequence of communications carried out between a set of clients and the server. The *projection* of a communication sequence σ on a client c, written $\sigma|_c$, is the sub-sequence of σ which contains only those communications initiated by client c. A computation of a RS-instance *induces* a communication sequence given by the sequence of actions along that computation.

It is important to distinguish between the case where a method is successfully

processed by the server, and where it is rejected without any server state change. This is done by mapping return codes to the abstract values $\{\mathsf{OK}, \mathsf{ERROR}\}$, where OK represents the first case and ERROR the second.

For a finite communication sequence σ, the set $assoc(\sigma)$ of resource identifiers defines those resources 'known' at the end of σ. For the empty communication sequence, $assoc(\lambda) = B$. Inductively, $assoc(\sigma; m)$ is $(assoc(\sigma) \cup L(m)) \backslash UL(m)$, if m has return code OK, and it is $assoc(\sigma)$, if the return code is ERROR.

For a finite computation with induced communication sequence σ, $assoc(\sigma)$ and $I \backslash assoc(\sigma)$ define the *associated* and *dissociated* resource identifiers, respectively. We associate a partial function $deref : I \mapsto 2^I \times D$ with the state of the server; $deref(i)$, if defined, is the current representation of the resource $\eta(i)$ (which must be defined if $deref(i)$ is defined).

4.2.3 Formalization of RESTful Behavior

For this section, fix a structure $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, and consider RS-instances. The two temporal properties discussed below define whether the behavior of an RS-instance is RESTful. It is usually more convenient to describe the failure cases, and also more helpful for the purpose of automatic verification. In the temporal formulas, we use a modified next-time operator, $\mathsf{X}_{\langle a \rangle}$, where a is an action. Its semantics is defined on a sequence with atomic propositions on each state and an action label on each transition. For a sequence σ and position i, define $\sigma, i \models \mathsf{X}_a(\varphi)$ to hold if $\sigma, i+1 \models \varphi$ and the transition from step i to step $i+1$ is labeled with a.

Before diving into the specifics, it is worthwhile to point out a couple of important considerations. First, as in any formalization of a hitherto informal concept,

there may be subtle differences between an informal idea and its formalization; we point out those that we are aware of. Second, a large part of the usefulness of a formalization lies in the testability of these properties. It is helpful to make a distinction between formal properties which can be tested given complete information of the implementation of clients and the server (a 'white-box' view), and those which can be tested only on the observable sequences of interaction between clients and the server (a 'black-box' view). The first viewpoint is interesting for model-checking; the second for run-time verification. Since we are targeting both approaches, we present the properties from both points of view, making it clear if one leads to a weaker test than the other. This distinction is important only for the safety and idempotence properties.

1. Stateless behavior. In ([Fie00], Chapter 5), this property is described as follows: *"... each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."* We formalize it by requiring that the server response to a request be functional; i.e., independent of client history or identity. (A 'client' should be understood to be a machine, rather than a user.) Failure of statelessness is shown by a finite computation followed by a two-way fork, where for some **distinct** client identifiers c, d, one branch of the fork contains the communication $c{::}op(i,a)/r_1(v_1)$, and the other branch contains the communication $d{::}op(i,a)/r_2(v_2)$, and either $r_1 \neq r_2$, or $v_1 \neq v_2$. This failure specification captures the situation where, given an identical history, the same method carried out by different clients has distinct results.

This is a branching-time property. The failure case is expressed as follows in a slight modification of Computation Tree Logic (CTL) [EC80], which allows the

operator $\mathsf{EX}_{\langle a \rangle}$, for an action a.

$$(\exists c, d\ \exists i, op, a, r_1, v_1, r_2, v_2 : c \neq d\ \wedge\ (r_1 \neq r_2\ \vee\ v_1 \neq v_2) \wedge$$
$$\mathsf{EF}(\mathsf{EX}_{\langle c::op(i,a)/r_1(v_1)\rangle}(true)\ \wedge\ \mathsf{EX}_{\langle d::op(i,a)/r_2(v_2)\rangle}(true)))$$

The property suffices to detect the common cases of hidden per-client state. One subtlety is that the the property is based on *observable*, semantic effects of a hidden state, not its syntactic presence. Hence, it holds of a server which retains auxiliary per-client information – such as a request counter – but does not use that information to influence the response to a request.

The formalization is also slightly stronger than the intended informal notion of statelessness, in the following sense. Consider a server which implements a method as "`if (client=c) then return 3 else return 4`". This has no hidden state, yet the method has different results for distinct clients c and d, and fails the property.

2. Hypertext-driven behavior. Informally, this property requires a client to access a resource only by 'navigating' to it from a root identifier. It is also referred to by the acronym HATEOAS, which stands for "Hypertext/Hypermedia As The Engine Of Application State". The failure specification is a finite computation with induced communication sequence of the form $\sigma; c::op(\ldots, i, \ldots)/rc(\ldots)$, for some σ, return code rc, method op, and resource identifier i among the arguments of op, such that all of the following hold: $i \notin assoc(\sigma|_c)$, and if L is the linked set of the last communication, then $i \notin L$. The return code and values are not important. It suffices that the identifier i is currently not associated from the perspective of the client c.

This condition can be expressed in Linear Temporal Logic (LTL) [Pnu77], most conveniently by using past temporal operators [LPZ85] to express the condition $i \notin \mathit{assoc}(\sigma|_c)$. The past-LTL formula for failure, denoted φ_{HT}, can be built up as shown below.

In the following, the predicate $\mathit{by}(m, c)$ is true if communication m is by client c; $\mathsf{OK}(m)$ is true if m has return code OK; $\mathit{arg}(m, i)$ is true if resource id i is an argument to the request in m; Y is the 1-step predecessor operator with variant $\mathsf{Y}_{\langle a \rangle}$ (formally, $\sigma, i \models \mathsf{Y}_{\langle a \rangle}(\varphi)$ if $(i \geq 1)$ and $\sigma, (i-1) \models \varphi$, and the transition from step i to step $i+1$ is labeled by a); and $p \mathbin{\mathsf{S}} q$ is the 'since' operator which holds if q holds in the past, and p holds since then. Precisely, $\sigma, i \models p \mathbin{\mathsf{S}} q$ **iff** $(\exists k : 0 \leq k \leq i : \sigma, k \models q \ \wedge \ (\forall j : k < j \ \wedge \ j \leq i : \sigma, j \models p))$. Note that $\neg Y(\mathit{true})$ is true only at the initial state of a sequence.

$$
\begin{aligned}
\varphi_{HT} = & (\exists c, i : \mathsf{F}(\mathit{access}(c,i) \ \wedge \ \neg \ \mathit{inassoc}(c,i))), \ \text{where} \\
\mathit{access}(c,i) = & (\exists m : \mathsf{X}_{\langle m \rangle}(\mathit{true}) \ \wedge \ \mathit{by}(m,c) \ \wedge \ \mathit{arg}(m,i) \ \wedge \ i \notin L(m)), \ \text{and} \\
\mathit{inassoc}(c,i) = & (\neg \ \mathit{revoked}(c,i)) \mathbin{\mathsf{S}} \mathit{granted}(c,i), \ \text{where} \\
\mathit{revoked}(c,i) = & (\exists m : \mathsf{Y}_{\langle m \rangle}(\mathit{true}) \ \wedge \ \mathsf{OK}(m) \ \wedge \ \mathit{by}(m,c) \ \wedge \ i \in \mathit{UL}(m)), \ \text{and} \\
\mathit{granted}(c,i) = & (\exists m : \mathsf{Y}_{\langle m \rangle}(\mathit{true}) \ \wedge \ \mathsf{OK}(m) \ \wedge \ \mathit{by}(m,c) \ \wedge \ i \in L(m)) \vee \\
& (\neg \mathsf{Y}(\mathit{true}) \ \wedge \ i \in B)
\end{aligned}
$$

An alternative view of this property: At the base of the hypertext-driven behavior property is the notion that all references made by clients to resource identifiers are based on prior familiarity with those identifiers. However, the description above does allow clients to become familiar with resource identifiers through the

explicit association of them (this is expressed as allowing access to a resource identifier that is in the linked set of the communication, which could mean in its request). This view may result in computations that, it could be argued, do not necessarily follow the intended principles of REST. For example, it allows for clients to 'guess' client identifiers that were associated by other clients. An alternative view of this property, therefore, would be to include in the linked sets of methods only resource identifiers that appear in responses, effectively banning any novel associations explicitly done by clients. Accounting in our formalism for such an interpretation would be relatively simple, and we leave it to the readers to do in the case that they wish to follow it.

3. Safety and idempotence. REST explicitly includes intermediaries in the model, such as caches and proxies. It is encouraged to have methods which are uniformly idempotent or safe, as intermediaries can more effectively use these methods to reduce latency or mask temporary server failures. While these properties are not required of REST methods, they can be formalized in LTL and model-checked. Unlike the two main properties, the formalization of safety and idempotence is different in the white-box and black-box views.

For the black-box setting, we require the following additional constructs. We suppose that there is a distinguished method, $\mathsf{READ}(i)$, where i is the target resource identifier. It returns either ERROR or $\mathsf{OK}(\mathit{deref}(i))$, the representation of the resource identified by i. The linked and unlinked sets are empty. We extend the equivalence relation '$\sim$' is to a list of return values: for lists a and b, $a \sim b$ holds if the lists have the same length and corresponding elements have the same types and are related by '$\sim$'. In the following, we also assume that one can identify whether a communication affects a resource; this information is typically available

for specific instances of REST, such as RESTful HTTP.

- **Safety of a method.** A method is considered safe if it does not modify resources. In the black-box view, changes to resources can be detected by means of READ methods. A failure for the safety of method *op* is a finite computation with communications c_1::$\mathsf{READ}(i)/\mathsf{OK}(r_1)$ and c_2::$\mathsf{READ}(i)/\mathsf{OK}(r_2)$ occurring in that order, with $r_1 \not\sim r_2$, where no intervening communication modifies or dissociates the resource $\eta(i)$ but includes at least one communication using *op*. Informally, failure of safety is signaled by a difference in the representation of the resource identified by i before and after method *op*.

- **Idempotence of a method.** For a method to be idempotent, repeated invocation should have no additional effect on resources. In the black-box view, such changes can be detected by means of READ methods. A failure for the idempotence of method *op* is a finite computation where the communications c_1::$op/rc(rv_1)$, c_2::$\mathsf{READ}(i)/\mathsf{OK}(r_1)$, c_3::$op/rc(rv_2)$, and c_4::$\mathsf{READ}(i)/\mathsf{OK}(r_2)$ occur in that order, $r_1 \not\sim r_2$ and the communications occurring between these distinguished ones do not dissociate i or modify the resource $\eta(i)$. Informally, the property detects failure by detecting a difference in the representation of a resource identified by i before and after the second instance of a communication with method *op*.

Both black-box properties are weaker than their white-box counterparts. For instance, it is possible that method *op* changes the server state of a resource – perhaps by incrementing an auxiliary counter – but this change is not propagated to the representation, and is hence unobservable by a READ. This violates safety in the white box view, but not in the black-box view.

4.2.3.1 Naming Independence We present an interesting consequence of the RESTful properties, which shows that the specific choice of naming function does not matter, if client-server behaviors are hypertext-driven. To make this precise, consider structures RS and RS' which are identical except for the naming functions. The naming functions, η and η', are required to map each base name to the same resource. The functions induce a name correspondence: a name i in an RS-instance corresponds to a name j in an RS'-instance if both map to the same resource, i.e., if $\eta(i) = \eta'(j)$.

If clients C_i and C'_i in the hypothesis of the theorem are based on the same program text, a sufficient condition for bisimularity up to naming is that names are used *opaquely*, i.e., no constant names are present, names can only be stored to and copied from variables, and the only relational test allowed for names is equality of name variables.

Theorem 4.1. *Consider an RS-instance M with clients $C_1, \ldots, C_k$ and server S, and an RS'-instance M' with clients $C'_1, \ldots, C'_k$ and server S'. Suppose that, for each i, clients C_i and C'_i are bisimular up to the naming correspondence, as are S and S'. Then, for each hypertext-driven computation σ of M, there is a hypertext-driven computation σ' of M' such that global states $\sigma(i)$ and $\sigma'(i)$ are bisimular, for each i, and the induced communication sequences match up to the naming correspondence.*

Proof: To simplify the notation in the proof, we will assume that the naming functions η and η' are onto the set of resources. Since these functions are injective on their domains, it follows that the partial function $\pi = \eta'^{-1} \circ \eta$ is a bijection on the domains of η and η' ($\pi : I \mapsto I$, such that for $i, j \in I$, $\pi(i) = j$ **iff** $\eta(i) = \eta'(j)$). The function π defines the equivalence between resource identifiers ('names'). (π

is extended naturally to sets of resource identifiers.)

We are given that clients C_i and C'_i are bisimular up to naming correspondence, as are the servers S and S'. This notion of bisimularity means that for related states (s, s'): any internal transition $s \xrightarrow{\tau} t$ has a matching transition $s' \xrightarrow{\tau} t'$ and (t, t') are related; and that for any communication transition $s \xrightarrow{m} t$, there is a transition $s' \xrightarrow{m'} t'$ such that (t, t') are related, m and m' have the same method and the same return code, the ids used in the requests of m and m' are related by π, and further, that $\pi(L(m)) = L(m')$ and $\pi(UL(m)) = UL(m')$.

Consider two global states to be bisimular if the local components in each are bisimular. Given a computation $\sigma = \sigma_0, \sigma_1, \ldots$ of M, we construct a computation $\sigma' = \sigma'_0, \sigma'_1, \ldots$ of M' such that, at each position k, the global states σ_k and σ'_k are bisimular, and the *assoc* sets for each client are identical up to name correspondence, i.e., for each i, if clients C_i and C'_i are identified by c_i and c'_i (correspondingly), $\pi(assoc(\sigma|_{c_i})) = assoc(\sigma|_{c'_i})$.

The construction is inductive: for the base case, $\sigma'_0 = \sigma_0$. Inductively, suppose that σ'_k is bisimular to σ_k. If σ_{k+1} is obtained by a local transition of some process, C_i, from σ_k, there is a corresponding local transition in C'_i which, when performed, results in a state σ'_{k+1} that is bisimular to σ_{k+1}, and no change in *assoc* sets. The other case is if σ_{k+1} is obtained by a communication m by some process, C_i, and S from σ_k. By bisimularity, there is a corresponding communication m' from C'_i and S' which, when performed, results in a state σ'_{k+1} that is bisimular to σ_{k+1}. Note that the linked and unlinked sets of m and m' are related by π. Following the definition of *assoc* and the inductive hypothesis, the *assoc* sets are also name equivalent for each client.

The construction ensures that, for any k, if the transition at step k in σ is by

client C_j and the request involves a resource id i, then step k of σ' is by client C'_j and uses the resource id $\pi(i)$. This, together with the name-equivalence of the *assoc* sets, ensures that the hypertext-driven property fails at step k of σ **iff** it fails at step k of σ'.

This, together with the inductively established correspondence between the states of σ and σ', establishes the claim. $\square$

4.3 REST on HTTP, and Variations

In this section, we show how the property templates from Section 4.2 can be instantiated for a concrete protocol, HTTP, which is the primary protocol used for constructing RESTful applications. The result is a formal definition of RESTful HTTP behavior.

4.3.1 A Formal HTTP Model

HTTP is a networking protocol for distributed, collaborative, hypermedia information systems [FGM+99]. The bulk of the interest in REST among developers is in the context of HTTP-based applications. We start by demonstrating how HTTP satisfies the framework requirements described in Subsection 4.2.1.

HTTP is typically used in a client-server model. HTTP resources are uniquely identified using their Uniform Resource Identifiers (URIs) [W3C05] (in the set I)[10]. For HTTP applications, the fields of a resource representation $\langle uris \in 2^I; d \in D \rangle$ are used as follows: *uris* is a set of URIs, *links* that exist in the resource, and d is any data, of any type, that is contained in a resource. It may include auxiliary data, such as counters, which is relevant to server-internal processes, but has no relevance

[10] The set of root URIs, B, is considered as 'common knowledge' for each specific web application (e.g.,'`www.thenation.com`'). Usually, $|B| = 1$.

to client behavior. Such data can be elided through an appropriate definition of '$\sim$'. The HTTP RFC [FGM+99] defines nine methods. We present here the four main methods, the remaining five have no impact on resources. To represent the HTTP concept of *subordinate* resources, we use a partial mapping, $S : I \mapsto 2^I$, which maps each resource identifier to the set of resource identifiers for its subordinate resources, if any. We only describe successfully processed communications, which return the abstract return code OK, all other codes map to ERROR. The main HTTP methods, with their linked and unlinked sets, are as follows.

- $\mathsf{GET}(i)/\mathsf{OK}(\mathit{deref}(i))$: The method returns the current entity (resource representation) of the resource identified by i from the server. Both L and UL are empty.

- $\mathsf{DELETE}(i)/\mathsf{OK}$: The method dissociates the resource identifier i on the server, resulting in $\mathit{deref}(i)$ bring undefined. Here, L is empty, and $UL = \{i\}$. The HTTP RFC actually only requires that the server 'intends' to dissociate it [FGM+99]. We discuss this more complex scenario in Subsection 4.3.3.

- $\mathsf{PUT}(i, \langle uris; d\rangle)/\mathsf{OK}$: The method associates a resource identified by i, if it is not already associated, and assigns a value to its corresponding entity so that $\mathit{deref}(i) = \langle uris; d\rangle$. If this is a new association, then $S(i) = \{\}$. Here, UL is empty, while $L = \{i\}$.

- $\mathsf{POST}(i, \langle uris; d\rangle)/\mathsf{OK}(j)$: The method associates a fresh resource, which is identified by j, and sets $S(j) = \{\}$ and $\mathit{deref}(j) = \langle uris; d\rangle$. The resource identified by j becomes a subordinate of the resource identified by i, and j is added to $S(i)$. Here, UL is empty, while $L = \{j\}$.

4.3.2 RESTful HTTP Properties

Using the HTTP modeling from Subsection 4.3.1, we can provide a more mathematical description of the properties from Subsection 4.2.3. These descriptions could be used to construct the automata from Subsection 4.4.3. (the description of HTTP stateless behavior could be used to refine the modified CTL formula from Subsection 4.2.3.) Clearly, if a HTTP-based application misuses the HTTP protocol or does not follow our model from Subsection 4.3.1, the following properties would not be useful for verifying that it is RESTful, and such a process would have to refer to another interpretation of the REST properties described in Subsection 4.2.3.

1. HTTP stateless behavior. The following describe the error cases.

In each case, there exist a finite communication sequence σ, $c_1, c_2 \in C$, and $i \in I$ such that at the end of some finite computation that induces σ:

- There exist $r_1, r_2 \in \langle 2^I \times D \rangle$, and a 'temporal fork' inducing the communication sequences

  $$\sigma; c_1{::}\mathsf{GET}(i)/\mathsf{OK}(r_1) \quad \text{and} \quad \sigma; c_2{::}\mathsf{GET}(i)/\mathsf{OK}(r_2)$$

  such that $r_1 \not\sim r_2$.

- There exist $r \in \langle 2^I \times D \rangle$, $j_1, j_2 \in I$, and a 'temporal fork' inducing the communication sequences

  $$\sigma; c_1{::}\mathsf{POST}(i, r)/\mathsf{OK}(j_1) \quad \text{and} \quad \sigma; c_2{::}\mathsf{POST}(i, r)/\mathsf{OK}(j_2)$$

such that $j_1 \neq j_2$.

- There exist $op \in \{\mathsf{GET}, \mathsf{DELETE}\}$, and a 'temporal fork' inducing the communication sequences:

$$\sigma; c_1{::}op(i)/\mathsf{OK}(\ldots) \quad \text{and} \quad \sigma; c_2{::}op(i)/\mathsf{ERROR}$$

- There exist $r \in \langle 2^I \times D \rangle$, and $op \in \{\mathsf{PUT}, \mathsf{POST}\}$, and a 'temporal fork' inducing the communication sequences:

$$\sigma; c_1{::}op(i, r)/\mathsf{OK}(\ldots) \quad \text{and} \quad \sigma; c_2{::}op(i, r)/\mathsf{ERROR}$$

(We mention no cases for methods DELETE or POST returning OK on both branches of a 'temporal fork', since these methods include no return values that could differ from each other.)

2. URI-driven behavior. This property, unlike the previous one, is stated in a positive form.

For all finite communication sequences σ, $c \in C$, $rc \in RETS$, $i, j, k \in I$, $d_1, d_2 \in D$, and $l_1, l_2 \in 2^I$, and for all finite computations that induce one of the following communications sequences

- $\sigma; c{::}\mathsf{GET}(i)/rc(\ldots)$
- $\sigma; c{::}\mathsf{DELETE}(i)/rc$
- $\sigma; c{::}\mathsf{POST}(j, \langle l_1; d_1 \rangle)/rc(\ldots)$ where $j = i$ or $i \in l_1$
- $\sigma; c{::}\mathsf{PUT}(j, \langle l_1; d_1 \rangle)/rc(\ldots)$ where $j \neq i$ and $i \in l_1$

at least one of the following holds:

- $i \in B$, and nowhere in $\sigma|_c$ was there a communication of the form $c::\mathsf{DELETE}(i)/\mathsf{OK}$
- $\sigma|_c$ contains a communication of the form

$$c{::}\mathsf{GET}(k)/\mathsf{OK}(\langle l_2; d_2\rangle)$$

  where $i \in l_2$, and nowhere in $\sigma|_c$ following this communication was there a communication of the form $c{::}\mathsf{DELETE}(i)/\mathsf{OK}$

- $\sigma|_c$ contains a communication of the form

$$c{::}\mathsf{POST}(k, \langle l_2; d_2\rangle)/\mathsf{OK}(i)$$

  where $i \notin l_2$, and nowhere in $\sigma|_c$ following this communication was there a communication of the form $c{::}\mathsf{DELETE}(i)/\mathsf{OK}$

- $\sigma|_c$ contains a communication of the form

$$c{::}\mathsf{PUT}(i, \langle l_2; d_2\rangle)/\mathsf{OK}$$

  where $i \notin l_2$, and nowhere in $\sigma|_c$ following this communication was there a communication of the form $c{::}\mathsf{DELETE}(i)/\mathsf{OK}$

(Note that communication sequences of the form $\sigma; c{::}\mathsf{PUT}(i, \ldots)/rc$ – for all $c \in C$ and $i \in I$ – are **always** allowed, even in applications that satisfy this property.)

Despite the fact that this property is not described here in its negated form, it could be translated into a monitoring automaton that identifies violating com-

munication sequences (the set of conditions out of which at least must hold above could be used to calculate, in an on-going way at any given state, the set *assoc* for the client. A violation may then occur for any of the three communications specified).

An alternative view of this property: As mentioned in the general REST description of this property in Subsection 4.2.3, an alternative reasonable interpretation of it would not allow clients to explicitly initiate novel associations of resource identifiers. In our HTTP model, as described in Subsection 4.3.1, this is only possible through PUT requests. To account for this alternative view, the property described above would have to include the following change: The fourth case, that handles PUT communications, would have to make sure that even the target resource of PUT communications was 'known' to the client already. Formally, that line in the definition would have to change to

$\sigma; c{::}\mathsf{PUT}(j, \langle l_1; d_1\rangle)/rc(\ldots)$ where $j = i$ and $i \in l_1$

(requiring that for such communication sequences, at least one of the requirements that follow would hold).

3. HTTP safety and idempotence. HTTP satisfies the following method-properties, which allows it to use caching efficiently and which makes it, therefore, a good candidate for implementing RESTful systems (if following the HTTP RFC [FGM⁺99], with methods' behavior as modeled in Subsection 4.3.1). Using the HTTP method GET instead of the abstract method READ from Subsection 4.2.3, described here are the error cases:

- **Safety of GET.** There exist a finite communication sequence σ, $c, c_1, c_2 \in C$, $i \in I$, $r_1, r_2 \in \langle 2^I \times D\rangle$, and there exists a finite computation that induces

the following communication sequence

$$\sigma; c_1{::}\mathsf{GET}(i)/\mathsf{OK}(r_1); \alpha_{\mathsf{GET}}; c_2{::}\mathsf{GET}(i)/\mathsf{OK}(r_2)$$

where $r_1 \not\sim r_2$.

α_{GET} represents any non-empty, finite, sequence containing at least one successfully processed GET communication. It must not contain any communications of the forms $c{::}\mathsf{DELETE}(i)/\mathsf{OK}$ or $c{::}\mathsf{PUT}(i, \ldots)/\mathsf{OK}$.

- **Idempotence of DELETE.** There exist a finite communication sequence σ, $c, c_1, c_2, c_3, c_4 \in C$, $i, j \in I$ $(i \neq j)$, $r_1, r_2 \in \langle 2^I \times D \rangle$, and there exists a finite computation that induces the following communication sequence

$$\sigma; c_1{::}\mathsf{DELETE}(i)/\mathsf{OK}; \alpha; c_2{::}\mathsf{GET}(j)/\mathsf{OK}(r_1);$$
$$\beta; c_3{::}\mathsf{DELETE}(i)/\mathsf{OK}; \gamma; c_4{::}\mathsf{GET}(j)/\mathsf{OK}(r_2)$$

where $r_1 \not\sim r_2$. We require that in such communication sequences the finite sequences β and γ do not contain communications of the forms $c{::}\mathsf{DELETE}(j)/\mathsf{OK}$ or $c{::}\mathsf{PUT}(j, \ldots)/\mathsf{OK}$.

- **Idempotence of PUT.** There exist a finite communication sequence σ, client identifiers $c, c_1, c_2, c_3, c_4 \in C$, $i, j \in I$, $r, r_1, r_2 \in \langle 2^I \times D \rangle$, and there exists a finite computation that induces the following communication sequence

$$\sigma; c_1{::}\mathsf{PUT}(i, r)/\mathsf{OK}; \alpha; c_2{::}\mathsf{GET}(j)/\mathsf{OK}(r_1);$$
$$\beta; c_3{::}\mathsf{PUT}(i, r)/\mathsf{OK}; \gamma; c_4{::}\mathsf{GET}(j)/\mathsf{OK}(r_2)$$

where $r_1 \not\sim r_2$. We require that in such communication sequences the fi-

nite sequences β and γ do not contain communications of the forms $c ::$ $\mathsf{DELETE}(j)/\mathsf{OK}$ or c::$\mathsf{PUT}(j, \ldots)/\mathsf{OK}$.

- All of the five HTTP operators that we do not mention in this chapter are safe (as well as idempotent).

The way in which the above method-properties should be considered, depends on whether a HTTP-based application follows the HTTP guidelines:

- In HTTP-based applications that are known to fully implement the HTTP RFC [FGM+99] as modeled in Subsection 4.3.1, the method-properties, as described here, are guaranteed to hold.

  Essentially, these properties provide a partial formal description for proper behavior of HTTP-based applications. These properties might be incorporated as part of a larger set of formal properties for the purpose of automatically verifying that systems follow the HTTP RFC.

- It is possible that a HTTP-based application misuses the HTTP communication protocol, but that the two REST requirements (statelessness and URI-driven behavior) can be guaranteed to hold. It is possible also, in such cases, to have idempotent or safe methods, but one must fall back on the general description from Subsection 4.2.3, as the effects of the HTTP methods may now be different from their standard effects. The effect that methods have on resource state is needed to properly specify the allowed communications in the failure cases (in the sub-sequences marked as α_{GET}, α, β, or γ above). If no information is available on the effects of methods for a given application, only a subset of failure cases can be properly defined (for instance, those where the sub-sequences such as α or β are empty). Given more (perhaps

partial) information on the effects of methods, the set of failure cases can be enlarged appropriately. A larger set of failure cases is clearly desirable for increasing the likelihood of identifying error cases.

4.3.3 Variations on RESTful HTTP Properties

In this section we present several common or reasonable modifications of the HTTP model from Subsection 4.3.1. We follow by pointing out the impact of these modifications on our RESTful HTTP properties from Subsection 4.3.2.

4.3.3.1 Cascade of DELETE Methods by Subordination As mentioned above, one side affect of the POST method is the creation of a subordination relation from the target resource identifier to the newly associated one. A common feature in many HTTP applications is the requirement that when a resource identifier is dissociated through a DELETE call, its subordinates are deleted as well (which, in turn, may trigger more dissociations of resource identifiers with higher degrees of subordination to the originally deleted one). In our model, this would translate into a modification to the linked set of DELETE communications.

We define a *subordination path from* i *to* j ($i, j \in I$) to be a finite sequence of resource identifiers, $i_1, i_2, \ldots, i_n$, such that $i_1 = i$, $i_n = j$, and for every $0 < t < n$, $i_{t+1} \in S(i_t)$ and $S(i_t)$ is defined.

This variation would have the following impact on the set of RESTful HTTP properties from Subsection 4.3.2:

- The URI-driven sequences property should be modified such that whenever we require that there is no communication of the form $c::\mathsf{DELETE}(i)/\mathsf{OK}$, we would also require that there would be no communications of the form

c::$\mathsf{DELETE}(q)/\mathsf{OK}$, for every $q \in I$ that has a subordination path from it to i.

- Safety of GET should be modified such that α_{GET} must not contain communications of the form $c :: \mathsf{DELETE}(j)/\mathsf{OK}$, for every $j \in I$ that has a subordination path from it to i.
- Idempotence of DELETE should include two modifications:
  - β and γ must not contain communications of the form c::$\mathsf{DELETE}(k)/\mathsf{OK}$, for every $k \in I$ that has a subordination path from it to j.
  - In the special case that there is a subordination path from i to j, we would expect both GET methods to return with the same error code (ERROR in our model).
- Idempotence of PUT should be modified such that β and γ must not contain communications of the form c::$\mathsf{DELETE}(k)/\mathsf{OK}$, for every $k \in I$ that has a subordination path from it to j.

4.3.3.2 Subordination Expressed as a Link A case to consider is that in which subordination is expressed as a link, i.e., for every $i \in I$ such that $\mathit{deref}(i) = \langle \mathit{uris}; d \rangle$, if $S(i)$ is defined then $S(i) \subseteq \mathit{uris}$. In this case, a side effect of the communication $c :: \mathsf{POST}(i, r)/\mathsf{OK}(j)$ would be the modification of the resource identified by i (to include j in uris).

This variation would have the following impact on the set of RESTful HTTP properties from Subsection 4.3.2:

- Safety of GET should be modified such that α_{GET} must not contain communications of the form c::$\mathsf{POST}(i, \ldots)/\mathsf{OK}(\ldots)$.

- Idempotence of DELETE should be modified such that β and γ must not contain communications of the form c::POST($j, \ldots$)/OK($\ldots$).

- Idempotence of PUT should be modified such that β and γ must not contain communications of the form c::POST($j, \ldots$)/OK($\ldots$).

4.3.3.3 Background Data Modifications by the Server In some cases, where the semantics of the domain D are such that it is is (partially or fully) dynamic by nature, HTTP allows the server to modify the data field of resource representations arbitrarily, in accordance with their semantics. An example is a 'current time' resource, whose value is updated by the server. Successive GET's on this resource would result in different values for the time, potentially violating the safety property of GET. This case can be handled by a proper definition of the data equivalence relation to ignore such changes.

4.3.3.4 Delayed Executions of Completed DELETE Communications In the HTTP RFC ([FGM+99]) it is said that when the server successfully processes a DELETE request it merely means that "at the time the response is given, it intends to delete the resource or move it to an inaccessible location".

Our interpretation of this quote from the HTTP RFC is that, instead of executing the dissociation immediately, the server only commits to doing so eventually, i.e., after some arbitrary, **yet finite**, delay. This means that the single resource identifier that is in the unlinked set of successfully processed DELETE communications is dissociated only after some arbitrary, finite, delay. One restriction that we have on this requirement is the following: For some resource identifier i, if at any point after a successfully processed DELETE(i) communication, but before i is dissociated, the server successfully processes a communication that has i in its linked

set, we waive the requirement that the server dissociates i (in fact, we forbid it, unless, naturally, a new $\mathsf{DELETE}(i)$ request arrives at the server). This additional restriction is not mentioned in the HTTP RFC, but since the whole description of DELETE is vague there, and since this interpretation is sensible, we assume it to be true.

It is worthwhile to note here that this interpretation of DELETE means that our inductive construction of the set $assoc(\sigma)$ (for any communication sequence σ) may no longer be helpful, due to the fact that there is no way of knowing when unlinked resource identifiers become dissociated. In the HTTP case of DELETE, as demonstrated below, this observation has no effect on our properties. However, it is possible that similar delays introduced to other, non-HTTP, systems would have to be approached differently because of their impact on the construction of $assoc(\sigma)$ (we do not consider such cases here).

This variation would have the following impact on the set of RESTful HTTP properties from Subsection 4.3.2:

- The HTTP Statelessness property should be modified to reflect the fact that a 'temporal fork' that describes two successfully processed DELETE communications initiated by different clients but with the same target resource identifier have identical effects on the system within any finite horizon. To express this, we have the additional failure cases given below.

  There exist finite communication sequences σ_1, σ_2, $c_1, c_2, c_3 \in C$, and $i_1, i_2 \in I$ such that at the end of some finite computation that induces σ_1 there is a 'temporal fork' and along its two branches, one of the following holds:

  - There exist $r_1, r_2 \in \langle 2^I \times D \rangle$, such that the communication sequences induced along the branches are

$\sigma_1; c_1{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}\mathsf{GET}(i_2)/\mathsf{OK}(r_1)$

and

$\sigma_1; c_2{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}\mathsf{GET}(i_2)/\mathsf{OK}(r_2)$

and $r_1 \not\sim r_2$.

- There exist $r \in \langle 2^I \times D \rangle$, $j_1, j_2 \in I$, such that the communication sequences induced along the branches are

  $\sigma_1; c_1{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}\mathsf{POST}(i_2, r)/\mathsf{OK}(j_1)$

  and

  $\sigma_1; c_2{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}\mathsf{POST}(i_2, r)/\mathsf{OK}(j_2)$

  and $j_1 \neq j_2$.

- There exist $op \in \{\mathsf{GET}, \mathsf{DELETE}\}$, such that the communication sequences induced along the branches are:

  $\sigma_1; c_1{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}op(i_2)/\mathsf{OK}(\ldots)$

  and

  $\sigma_1; c_2{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}op(i_2)/\mathsf{ERROR}$

- There exist $r \in \langle 2^I \times D \rangle$, and $op \in \{\mathsf{PUT}, \mathsf{POST}\}$, such that the communication sequences induced along the branches are:

  $\sigma_1; c_1{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}op(i_2, r)/\mathsf{OK}(\ldots)$

  and

  $\sigma_1; c_2{::}\mathsf{DELETE}(i_1)/\mathsf{OK}; \sigma_2; c_3{::}op(i_2, r)/\mathsf{ERROR}$

- Safety of GET should be modified such that σ must not contain any successfully processed $\mathsf{DELETE}(i)$ communications, unless it also contains a successfully processed $\mathsf{PUT}(i)$, or a $\mathsf{POST}(\ldots)/\mathsf{OK}(i)$, after the latest such occurrence.

- Idempotence of DELETE should be modified such that σ and α must not contain communications of the form $c :: \mathsf{DELETE}(j)/\mathsf{OK}$, unless they also contain a successfully processed $\mathsf{PUT}(j)$, or a $\mathsf{POST}(\ldots)/\mathsf{OK}(j)$, after the latest such occurrence.

- Idempotence of PUT should be modified such that σ and α must not contain communications of the form $c{::}\mathsf{DELETE}(j)/\mathsf{OK}$, unless they also contain a successfully processed $\mathsf{PUT}(j)$, or a $\mathsf{POST}(\ldots)/\mathsf{OK}(j)$, after the latest such occurrence.

4.3.4 Distinguishing REST from HTTP

Following are some interesting hypothetical applications which clarify the differences between HTTP and REST, and which address some common misunderstandings regarding RESTful HTTP.

Consider an application which uses only two HTTP methods: PUT and GET. A client encodes methods in the *uri* argument of $\mathsf{PUT}(uri, junk)$ requests, where $junk$ - a resource representation - is a meaningless constant. A GET communication is used by a client to examine the state of the server. This application is compliant with the HTTP RFC, as there is no restriction on the PUT communications' return values. However, it is non-RESTful, since it would either have to include an infinite set of root identifiers (each *uri* argument being one), or it would violate the hypertext-driven behavior property. The Flickr API is non-RESTful for a similar reason.

Consider an application which relies entirely on POST communications, and uses a single root identifier, *base*, for all such communications (one may consider $B = \{base\}$). In any $\mathsf{POST}(base, \langle base; data\rangle)/\mathsf{OK}(uri)$ communication, clients

encode methods in the *data* field. We consider two variants:

1. The return value of an method is encoded in the newly associated URI *uri*, returned as a result of POST. This is compliant with the HTTP RFC, but it goes against the notion of dividing information into distinct resources, as the base URI must be treated as a single resource. As there is no division into resources (which would be created by – and used to identify – different clients), this application is likely to violate the HTTP statelessness property. Moreover, it is also likely to violate the resource identifier opaqueness assumption from Subsection 4.2.3.1, as a program must interpret the URI strings returned by POST. While the opaqueness assumption is not an essential part of REST, it is important to simplify program development and maintenance.

2. The newly associated URI *uri* is used to point to a resource whose representation is the result of the method, and which is later retrieved by a GET on the *uri*. This violates the HTTP RFC, which requires that the result of POST identifies a resource with the supplied data as its representation. As in the previous case, this application is also likely to violate the HTTP statelessness property.

4.4 Automated Verification of RESTful Behavior

In this section, we formulate and discuss questions relevant to the automated verification of RESTful behavior. We give preliminary results and point to questions that are still open.

4.4.1 Computation Model

The somewhat informal model used previously can be made precise as follows. Client and server processes are modeled as labeled transition systems. A communication is modeled as a CCS synchronization [Mil89]. Hence, in a communication of the form 'request/response', a client offers this communication at its state, the server offers to accept it, and the two are synchronized to effect the communication. Processes may have internal actions, including internal non-determinism. The CCS model is appealing for its simplicity but assumes atomic communication. We formulate problems and solutions in this model. Subsequently, we discuss how the atomicity requirement may be relaxed, which brings the analysis closer to real implementation practice.

4.4.2 Fundamental Questions

The two properties of REST, statelessness and hypertext-driven behavior, lead to the following key verification questions.

ST Does a client-server application M satisfy the statelessness property?

HT1 For a client-server application M, does its specification, φ, hold for all computations where client behavior is hypertext-driven?

HT2 For a client-server application M, do all non-hypertext-driven computations satisfy a 'safe-behavior' property ξ?

These fundamental questions may be asked for a program with a fixed set of clients and resources, or in the parameterized sense. One may also ask if violations of these properties can be detected using run-time monitors. Another interesting

question is whether, given an application specification, one can synthesize a server which satisfies it (again, fixed or parameterized).

4.4.3 Automata Constructions

A nondeterministic automaton which detects a *failure* of the hypertext-driven behavior property works as follows. For a given input word, the automaton guesses the client and resource identifier with which to instantiate the failure specification, then keeps track of whether the resource id belongs to the current *assoc* for that client. It accepts if, at some point, there is a request by the client using the resource id, but the id is not part of the current *assoc* set. Keeping track of whether a resource id belongs to the *assoc* set for a client does not require computing the *assoc* set. A simple two-state machine suffices, with states $In(c, i)$ and $Out(c, i)$. If the current communication m is by client c and is successful, a transition is made from $In(c, i)$ to $Out(c, i)$ if $i \in UL(m)$, and from $Out(c, i)$ to $In(c, i)$ if $i \in L(m)$. Otherwise, the state is unchanged. The number of automaton states, therefore, is polynomial in $|I|$ and $|C|$.

The deterministic form of this automaton must track all clients and resource ids simultaneously. Thus, the size of a state of the deterministic automaton is $O(|I| \cdot |C|)$, and its state space is exponential: $O(2^{|I| \cdot |C|})$.

The non-deterministic automaton for the failure of safety properties guesses a resource identifier (i), as well as identities of two clients that would send READ requests, with which to instantiate the failure specification. It then guesses a location of a $\mathsf{READ}(i)$ communication, stores its return value, and checks for a sequence of allowed communications and with the candidate method followed by another $\mathsf{READ}(i)$ communication. At this point the automaton compares the stored

value with the second returned one, accepting if the two are not related by '$\sim$'. The size of an individual state in this automaton is, therefore, $O(\log(|I|) + \log(|C|) + \log(|R|))$, where R is the set of possible return values. The failure automaton for idempotence has more guesses to make, but follows a similar structure.

4.4.4 Model-Checking for Fixed Instances

A fixed instance has a fixed set of resources and clients. The parameters of interest are the sets in the underlying resource structure: the clients, C, the resource identifiers, I, and the data domain, D.

Statelessness is expressed in a slight variant of CTL, as described previously. (The extension does not affect model-checking complexity.) The indexed property expands out to a propositional formula which is polynomial in the sizes of I and D. Hence, using standard CTL model-checking algorithms [CE81, QS82], the ST property can be verified in time linear in the overall application state space and polynomial in the resource structure parameters.

Property HT1 can be verified as follows. A violation of HT1 is witnessed by a computation where all clients are hypertext-driven but φ is false. This can be checked using automata-theoretic model checking [VW86] by forming the product of the application process with (1) a Büchi automaton for the negation of φ, and (2) an automaton which checks that all clients follow hypertext-driven behavior. The property is verified **iff** the product has an empty language. The second automaton is the deterministic automaton from Subsection 4.4.3, with negated acceptance condition.

Property HT2 can be verified by forming the product of the application process with (1) a Büchi automaton for the negation of ξ, and (2) an automaton which

checks for failure of hypertext-driven behavior by some client. The property is verified **iff** the product has an empty language. The second automaton is the non-deterministic failure automaton from Subsection 4.4.3. The verification takes polynomial time if the size of the application state space is polynomial in the parameter sizes. The verification of HT1 is significantly more difficult.

Theorem 4.2. *Verification of HT1 for a fixed instance is PSPACE-hard in the number of resources. It is in PSPACE if a state of the application and of the negated specification automaton can be described in space polynomial in the parameter sizes.*

Proof Sketch: Membership in PSPACE is straightforward, by observing that the automaton used to describe the hypertext-driven property for HT1 has a state size which is polynomial in the the parameter sizes.

PSPACE-hardness for HT1 holds under severe restrictions: a single client, where client, server, and negated specification automaton have a state-space with size polynomial in the parameters' sizes. The reduction is from the question of deciding, given a Turing Machine (TM) M and input x, whether M accepts x within the first $|x|+1$ tape cells, which is a PSPACE-complete problem (IN-PLACE ACCEPTANCE in [Pap94]). The reduction uses the server state to store the TM head position, while a TM configuration is encoded in the implicitly defined *assoc* set for the client, using resources to represent tape cell contents.

Proof: In this part, we give a proof of the PSPACE-hardness of HT1.

The result is based on a reduction from the PSPACE-complete problem called IN-PLACE ACCEPTANCE in [Pap94] (Chapter 19, Section 19.3). This is the problem where, given a Turing Machine and an input string x, one has to determine whether the TM accepts x without leaving the first $|x|+1$ tape cells.

Given a deterministic TM M and input x, we construct a server process S, a client process C, and a collection of resources. The number of resources, as well as the state spaces of C and S, is polynomial in the description of the TM and x. The set of resources, and the results of methods on them is as follows

1. For each position i from 0 to $|x|$ and each symbol a, there is a cell-value resource named `tm/head/i/value/a`

2. There are two *root resources*: `tm/head` and `tm/initial`.

The server, S, works as follows. It keeps track of the head position, which requires $|x| + 1$ states, and responds to methods on the resources as shown below. It is easy to check that the server is stateless in the REST sense.

The server *does not* keep track of the tape contents, as that would require an exponential number of states. Instead, the tape contents are implicitly represented at any stage by the current *assoc* set for the client. The computation of M on x is simulated by a hypertext-driven computation of the client-server application.

1. `GET tm/initial` returns the list of resources of the form `tm/head/i/value/a` representing the initial tape contents, x. This is a constant list.

2. `GET tm/head` returns the head position (the state of the server)

3. For `PUT tm/head k`, if the position k is out of bounds (i.e., less than 0 or greater than $|x|$) the server enters a special REJECT state. Otherwise, the state is changed to represent position k. The response is OK.

4. For all valid values of i and a, `GET /tm/head/i/value/a` and `DELETE /tm/head/i/value/a` are accepted with return code OK.

In the description, we use resource templates explicitly for clarity and simplicity. In no way, however, does the operation of the client depend on the particular choice of names, since these can be replaced by URIs pointed to by tags.

The client, C, works in phases. It maintains the following invariant: for any finite computation reaching the start of the k'th phase, if the computation is hypertext-driven, the list of cell value resources in its *assoc* set is identical to the TM configuration after k steps, and the state of the server is the head position in that configuration. The client process uses internal non-determinism in each phase, but the structure of a phase ensures that at the end of a phase, only one of the non-deterministically defined runs is hypertext-driven. Thus, while the program as a whole generates many runs, only one is hypertext-driven.

The client state includes the control state of the TM, a constant number of TM symbols, and a constant number of integer values in the range 0 to $|x|$. The size of the state space is thus polynomial in the input size. To each state is attached a constant table of all value resource ids (i.e., those of the shape `tm/head/i/value/a`) indexed by the pair (i, a). As this constant lookup table is built into every state, it does not contribute to the size of the space.

The pre-phase operation of the client is to invoke `GET tm/initial`. The set of cell value resources returned by the server, and made part of the client's *assoc* set, is the set corresponding to the initial TM configuration. The server state is 0. This establishes the invariant for the start of the first phase. Each phase goes through the following steps.

1. The client halts if the TM control state is accepting. Otherwise, it continues as follows.

2. The client does `GET tm/head`. The server responds with an integer i, which

is the current head position. This is stored at the client.

3. The client chooses a symbol non-deterministically and stores the choice. For the choice, say a:

   (a) The client obtains the stored table entry for (i, a), and invokes `GET tm/head/i/value/a`.

   By the invariant, the wrong choice of a — i.e., a value not in the TM configuration at position i — forces a violation of the hypertext-driven property, as the resource id for this invocation is not in the *assoc* set.

   (b) The client uses the stored TM control state to compute the new value at the i'th position, the position increment, and the new control state. Let b be the new value and d the increment. The client issues the request `PUT tm/head (i+d)`, followed by `DELETE tm/head/i/value/a` and then `PUT tm/head/i/value/b`.

   The first PUT updates the head position, the second removes the URI for (i, a) from the assoc set, and the last PUT ensures that the URI for the pair (i, b) enters the assoc set. These actions re-establish the invariant.

Multiple runs are generated within a phase due to the non-deterministic choice of symbol. However, only one run (for the correct choice) is hypertext-driven. Since hypertext-driven behavior is a safety property, all extensions of the non-hypertext-driven runs also fail the property.

From the invariant, it follows that the TM accepts x within the first $|x|+1$ positions if, and only if, the (unique) hypertext-driven computation of the client halts and the server never enters its REJECT state. Coupled with the PSPACE-hardness

of the IN-PLACE ACCEPTANCE condition, this shows that the question of determining if all hypertext-driven computations satisfy a temporal property is also PSPACE-hard, even for simple properties (in this case $\mathsf{F}(HALT) \wedge \mathsf{G}(\neg REJECT)$).

□

4.4.5 Parameterized verification

The parameterized verification question has particular importance, as web applications usually handle a large number of clients and resources. Since statelessness is not a given, it is necessary to assume a server which stores information about each client, which implies that the state space of the server is also unbounded. Nonetheless, the problem can be solved under certain assumptions.

Suppose that clients have a finite state space, X, and that the state space of the server can be written as $Y \times [C \to Z]$, where Y and Z are finite sets. Thus, a global state of an instance with N clients is a triplet (c, a, b), where c is an array of client states, of size N, a is the finite part of the server state, and b is an array of N server-side entries. Assume further that on receiving a request from client i, the server update depends only on, and may only modify, the components a and $b(i)$; i.e., the new entry for client i does not depend on the entries of the other clients. Then, by a change of viewpoint, one may combine the entry $b(i)$ on the server with the state $c(i)$ of client i, obtaining an equivalent application where the new client space is $X \times Z$, and the server space is Y. Both spaces are now finite, although there is still an unbounded number of clients. This situation fits the model in [GS92], where an algorithm is given for checking linear-temporal properties. The algorithm has very high worst-case complexity, however, so it may be more fruitful to try alternative methods, such as the method of invisible

invariants [PRZ01, Nam07], or methods based on upward-closed sets [ACJT96].

Several questions remain open. The modeling above implicitly assumes a bounded set of resources and data values. Moreover, the suggested algorithm applies only to linear-time properties and cannot, therefore, be used to check statelessness.

4.4.6 Run-Time Monitoring

Perhaps the most promising immediate application of the formalization is run-time monitoring. In this setting, the client-server communications are captured by an intermediate proxy, which passes them through analysis automata. This method can be applied to the properties HT1 and HT2; statelessness, being a branching-time property, cannot be checked at run-time, unless some form of backtracking is implemented. The automata described in Subsection 4.4.4 for model-checking HT1 and HT2 can be used for run-time verification of safety specifications. The non-deterministic automata used for checking hypertext-driven behavior must be determinized for run-time analysis. This can be done on the fly, as is the case for implementations of the Unix `grep` command (cf. [ALSU07]). The size of the deterministic automaton state is $O(|I| \cdot |C|)$, so the required storage is $O(|I| \cdot |C| \cdot K)$, where K is the state-size of the negated specification automaton. For each communication, the update of the automaton state requires time proportional to the size of the state, and is hence polynomial in the resource parameters. An alternative to run-time verification is off-line testing of a logged communication sequence.

4.4.7 Synthesizing Servers

A particularly intriguing question is the possibility of synthesizing RESTful servers. A specific question is the following: given a resource structure and a specification φ, synthesize a **stateless** server which satisfies φ. We show below that, under certain assumptions, the statelessness constraint can be dropped.

We define a server specification φ to be universally synthesizable if there exists a server implementation which satisfies φ given any set of clients. A sufficient condition for φ to be universally synthesizable is if it is insensitive to client ids and is synthesizable for a single, arbitrary client. Insensitivity means that for every two communication sequences σ, δ which agree up to client ids in communications, $\sigma \models \varphi$ **iff** $\delta \models \varphi$.

Lemma 4.1. *For any φ that is insensitive to client ids, φ is synthesizable with client ids set $\{c_{dummy}\}$* **iff** *φ is universally synthesizable.*

Proof: The right-to-left direction of the proof is trivial.

For the other direction, let M be the machine synthesized for $\{c_{dummy}\}$ and φ. Now consider computations of M given any set of client ids C. Let α be a communication sequence that is induced by such a computation. We have to show that $\alpha \models \varphi$. Let β be obtained from α by replacing all client ids in α with c_{dummy} (note that this requires c_{dummy} to send arbitrary request). By the synthesis process for M, we know that $\beta \models \varphi$. As α and β agree up to client ids, and since φ is insensitive to client ids, it follows that $\alpha \models \varphi$. $\square$

Theorem 4.3. *Consider a server temporal logic specification φ. The specification φ is deterministically and universally synthesizable* **iff** *$ST \wedge \varphi$ is deterministically and universally synthesizable.*

Proof: Clearly, one side of the proof (assuming that $ST \wedge \varphi$ is deterministically and universally synthesizable) is trivial.

Assuming that φ is deterministically and universally synthesizable, we prove that $ST \wedge \varphi$ is deterministically and universally synthesizable. Let M be a deterministic application (server) that satisfies φ given any set of client identifiers C. We construct M' from M such that M' is deterministic, stateless and satisfies φ.

M' is defined as a pipeline of components. The first component, IN, rewrites incoming requests by (1) changing the client id of the request to a constant id, $c_{dummy} \notin C$, and (2) adding a fresh Unique Identifier (UID) as an additional argument. The modified request is sent to a variant $\hat{M}$ of M, which processes the request as M would, ignoring the request UID, but that places that UID into the response as an additional argument. The component IN uses the UIDs of responses to direct them to the clients making the requests (after removing the UIDs).

M' operates deterministically by construction. We now show that any computation of a set of clients with M' satisfies φ. Consider such a computation, σ. This induces a computation δ at the interface between IN and $\hat{M}$, where requests and responses are rewritten as described previously. Erasing the request identifiers from σ, we obtain a sequence, say γ, of M over the single client c_{dummy}. This satisfies φ by construction of M. As φ does not depend on request identifiers, δ satisfies φ. This implies that σ satisfies φ, as the property is insensitive to client identifiers.

The proof that M' is stateless is by contradiction. Say there exists some finite communication sequence, σ, and two distinct client identifiers $c, d \in C$ such that M' generates the communication sequences $\sigma; c :: op(i,a)/r_1(v_1)$, and

$\sigma; d{::}op(i,a)/r_2(v_2)$, where either $r_1 \neq r_2$, or $v_1 \neq v_2$ (op, i, and a being elements of the appropriate sets, and everything else being equal). By the definition of M', this computation induces a computation at the inner M where the communications from c, d on the fork are rewritten to originate from c_{dummy}, creating identical requests. The differing responses imply that M is non-deterministic, which is a contradiction to the assumption. □

The synthesis problem for LTL specifications, assuming a bounded state-space, was solved in [PR89b]. Implementing the intermediary adds constant complexity. Some subsets of LTL have polynomial synthesis algorithms [PPS06, PP06].

Adding the assumption that client interactions are hypertext-driven may make an otherwise-unsynthesizable specification synthesizable, but it may also add significantly to the specification complexity (by expanding its representation). The question of synthesizing unbounded state-space specifications (e.g., without limiting the number of clients) is still open.

4.4.8 Relaxing The Atomicity of Communications

So far, we have assumed that communications are atomic. In real implementations, however, a request and its response are distinct actions. This allows requests from different clients to overlap in time. To handle this concurrency, we assume that the server is linearizable [HW90]. Every computation produces results which are equivalent to one where each method takes effect atomically.

Hypertext-driven behavior is formulated entirely in terms of the request and response parameters. If clients are not allowed to issue concurrent requests, then hypertext-driven behavior holds of a computation **iff** it holds of its linearization. Assume that the service specification is also defined on communication sequences,

and has the same property. Then, it suffices to check properties over the linearized subset of computations, which corresponds to the atomic communication model. This reasoning does not apply to statelessness, which is a branching property, and thus outside the scope of linearizability. Further work is necessary to formulate a notion like linearizability for branching-time properties.

4.5 Related Work and Conclusions

There is surprisingly little in the literature on formal definitions and analysis of REST. In [HG10], the authors describe a pi-calculus model of RESTful HTTP. This model, however, comes across as a mechanism for programming a specific type of RESTful HTTP application. The paper does not consider the general properties of REST: statelessness and hypertext-following, nor does it describe a methodology for checking that arbitrary implementations satisfy these properties. There are also a number of books and expository articles on REST, but those do not include formal specifications, nor do they consider analysis questions.

Our work appears to be – to the best of our knowledge – the first to precisely formulate the key properties of REST, and to demonstrate interesting consequences, such as naming independence and the PSPACE-hardness of verification. This work also opens up a number of interesting questions. One is to use the formalization as a basis to investigate questions about REST itself: for instance, how to combine authentication with REST, and how to extend REST to executable representations [EGST07]. We have argued that the parameterized model-checking and synthesis questions are especially relevant for web applications using REST. Constructing a practically usable verifier for REST properties is itself a non-trivial task. We have experimented with simple examples verified using SPIN [Hol03]. An

effort to use JPF [VHB+03] to verify applications written in the JAX-RS extension of Java was unsuccessful, however, as JPF currently lacks support for key libraries in JAX-RS. Our current focus is on creating a run-time checker, which has the advantage of being independent of implementation language.

To summarize, the formal modeling of REST clarifies its definition, and also raises several challenging questions, both in modeling and in automated analysis.

Chapter 5

Summary

This dissertation concerns with two directions for the utilization of formalizations in the form of temporal logics for the purpose of insuring the 'correctness' of systems.

Chapter 2 and Chapter 3 presented the notion of synthesis and approached the subjects of 'effective' synthesis of synchronous systems, as well as synthesis of asynchronous synthesis with multiple variables. Chapter 4 took the other direction, and rather than deal with synthesis presented a way to use formal expression of desired systems' properties for the task of verifying RESTfulness.

In Chapter 2, we described a syntactic reduction that was originally presented in [PPS06], and the fact that it fails to present a complete solution for the problem of synthesizing synchronous systems by generating 'false negatives' at times: Declaring that some synchronously realizable specifications are unrealizable. We followed by analyzing such false classifications, and proved what problem is really solved by the syntactic reduction. We then described a class of LTL specifications that are guaranteed to never cause the syntactic reduction to produce wrong results – those

that contain well-separated environments, as well as an effective algorithm for the identification of such specifications without increasing the complexity of the overall synthesis process. Finally, we described a complexity-preserving methodology for the correct handling of specifications without well-separated environments, which allows for the syntactic reduction to provide a sound and **complete** solution for the problem of effective synchronous synthesis of LTL specification with $GR(1)$ winning conditions. Considering the class of specifications with well-separated environments, we discussed what kind of specifications could be excluded from it in order to provide a 'tight' description of the problematic specifications. One interesting question that arises of this work would be the identification of the exact set of specification that cause false classification.

Chapter 3, on the other hand, concerns with the problem of synthesizing asynchronous systems from LTL specifications. We start by presenting two models of asynchronous interactions of systems with multiple inputs and outputs: by-demand and round-robin, and continue with the more canonical one after proving that the two are equally expressive. After presenting the Rosner reduction from [PR89b], we expand it to the case of multiple variables, and describe the kernel formula $\mathcal{X}^{n,m}$ that is constructed from the specification φ and for which we prove that φ is asynchronously realizable **iff** $\mathcal{X}^{n,m}$ synchronously realizable. This result provides a sound and complete solution for the problem of synthesizing asynchronous systems from LTL specifications, by reducing it to the problem of synthesizing synchronous systems that could be handled according to the algorithm provided in [PR89a].

To address the fact that handling $\mathcal{X}^{n,m}$ is an extremely complex, practically unfeasible, task, we continue by suggesting several heuristics for the effective handling of it in restricted cases. We develop an over-approximation of $\mathcal{X}^{n,m}$, and an

under-approximation of it, that are used as two-way 'bounds' of it; If the former is synchronously unrealizable, then the underlying specification φ is asynchronously unrealizable and if the latter is synchronously realizable, then φ is asynchronously realizable and we follow by describing an effective algorithm for synthesizing it. In order to prove asynchronously realizability, we introduce a few constraints on φ, the most notable of those are its limitation to the case of single-input systems and the requirement of identifying for it an asynchronous strengthening ψ. Both the over-approximation and under-approximation of $\mathcal{X}^{n,m}$ could be effectively tested for synchronous realizability, unlike $\mathcal{X}^{n,m}$, producing the desired effective solution to the asynchronous problem.

Expanding the set of specifications that could be effectively tested for asynchronous realizability would be one interesting extension of this work, as well as the addition of methods for the minimization, and optimization, of the synthesized systems.

Chapter 4 concerns with verification as an application of temporal logic formalization, and not with synthesis. After introducing the fundamental objective of REST, as well as the problem of its frequent misunderstanding, we described a formal framework for the description of client-server applications, for the purpose of formalizing REST. We followed by defining the two key REST properties – statelessness and hypertext-driven behavior – both in temporal logic, as well as methods properties that maximize reliance on caching – safety and idempotence. These method properties were formalized using temporal logic as well.

We used our formalized definition to describe how to verify RESTfulness using model-checking ([CE81, QS82]), and analyzed the complexity of such a process. We further used our formal model to address several other useful verification tech-

niques, including the verification of parameterized systems and the verification of 'black box' systems using run-time monitoring. Finally, we proved the the task of synthesizing stateless servers could be reduced, in linear time, to that of general synthesis of servers.

We hope that this work would alleviate some of the challenges that developers encounter today when trying to understand the principles of REST. Additionally, we believe that it presents an interesting and challenging test-case for further work in the field of verification. For one, it lays the foundations for developing efficient tools for the 'certification' of systems as RESTful.

Bibliography

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay. General decidability theorems for infinite-state systems. In *LICS*, 1996. (Cited on page 150.)

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991. (Cited on page 9.)

[ALSU07] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools, Second Edition.* Addison Wesley, 2007. (Cited on page 150.)

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998. (Cited on pages 6 and 7.)

[AT04] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004. (Cited on pages 6 and 7.)

[BGHJ09] R. Bloem, K. Greimel, T.A. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *Proc. 9th Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'09), Austin, Texas*, pages 85–92, 2009. (Cited on page 19.)

[BGJ$^+$07a] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Design Automation and Test in Europe*, pages 1188–1193, 2007. (Cited on pages 8 and 41.)

[BGJ$^+$07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, volume 190 of *Electronic Notes in Computer Science*, pages 3–16, 2007. (Cited on pages 8 and 41.)

[BHIBL08] C. Bizer, T. Heath, K. Idehen, and T. Berners-Lee. Linked data on the web (LDOW2008). In *WWW*, pages 1265–1266, 2008. Talk by Tim Berners-Lee at TED 2009: `http://www.w3.org/2009/Talks/0204-ted-tbl/`. (Cited on page 116.)

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969. (Cited on pages 5 and 40.)

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp.*

Sci., pages 52–71. Springer-Verlag, 1981. (Cited on pages 5, 115, 144, and 158.)

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 8:244–263, 1986. (Cited on page 10.)

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999. (Cited on pages 2 and 106.)

[CHJ08] K. Chatterjee, T.A. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *19th International Conference on Concurrency Theory (CONCUR03)*, volume 5201 of *Lect. Notes in Comp. Sci.*, pages 141–161. Springer-Verlag, 2008. (Cited on page 23.)

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963. (Cited on pages 5 and 40.)

[CKGC+07] D.C. Conner, H. Kress-Gazit, H. Choset, A. Rizzi, and G.J. Pappas. Valet parking without a valet. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 572–577. IEEE, 2007. (Cited on pages 8 and 41.)

[DBPU10] N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behavior models. In *18th International Symposium on Foundations of Software Engineering*, Santa Fe, NM, USA, 2010. ACM. (Cited on page 41.)

[DBPU11] =N. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. Synthesis of live behavior models for fallible domains. In *33rd International*

Conference on Software Engineering, Waikiki, HI, USA, May 2011. ACM. (Cited on page 41.)

[DH99] W. Damm and D. Harel. LSC's : Breathing life into message sequence charts. In P.Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 293–312. Kluwer Academic Publishers, 1999. (Cited on page 2.)

[EC80] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Int. Colloq. Aut. Lang. Prog.*, volume 85 of *Lect. Notes in Comp. Sci.*, pages 169–181. Springer-Verlag, 1980. (Cited on pages 1 and 120.)

[EGST07] J. R. Erenkrantz, M. M. Gorlick, G. Suryanarayana, and R. N. Taylor. From representations to computations: the evolution of web architectures. In *ESEC/SIGSOFT FSE*, pages 255–264, 2007. (Cited on page 154.)

[EH86] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'not never' revisited: On branching time versus linear time. *J. ACM*, 33:151–178, 1986. (Cited on page 10.)

[ES84] E.A. Emerson and A.P. Sistla. Deciding full branching time logic. *Inf. and Cont.*, 61:175–201, 1984. (Cited on page 10.)

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. W3C RFC 2616, June 1999. `http://www.w3.`

org/Protocols/rfc2616/rfc2616.html. (Cited on pages 114, 127, 128, 132, 134, and 137.)

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irving, 2000. (Cited on pages 113, 115, and 120.)

[Fie08a] R. T. Fielding. http://roy.gbiv.com/untangled/2008/no-rest-in-cmis#comment-697, 2008. (Cited on page 114.)

[Fie08b] R. T. Fielding. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008. (Cited on page 114.)

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992. (Cited on page 149.)

[HG10] A. G. Hernández and M. N. Moreno García. A formal definition of RESTful semantic web services. In *WS-REST*, pages 39–45, 2010. (Cited on page 154.)

[Hol03] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003. Also see http://spinroot.com. (Cited on page 154.)

[HP06] T.A. Henzinger and N. Piterman. Solving games without determinization. volume 4207 of *Lect. Notes in Comp. Sci.*, pages 394–410. Springer-Verlag, 2006. (Cited on page 41.)

[HT87] T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th Int. Colloq. Aut. Lang. Prog.*, volume 267 of *Lect. Notes in Comp. Sci.*, pages 269–279. Springer-Verlag, 1987. (Cited on page 10.)

[HW90] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. (Cited on page 153.)

[KGFP07a] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. From structured english to robot motion. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2717–2722. IEEE, 2007. (Cited on page 8.)

[KGFP07b] H. Kress-Gazit, G.E. Fainekos, and G.J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3116–3121. IEEE, 2007. (Cited on pages 8 and 41.)

[KN11a] U. Klein and K. S. Namjoshi. Formalization and Automated Verification of RESTful Behavior. In *G. Gopalakrishnan and S. Qadeer, editors* Proc. 23rd Intl. Conference on Computer Aided Verification (CAV'11), volume 6806 of Lect. Notes in Comp. Sci., pages 541–556. Springer-Verlag, 2011. (Cited on page 115.)

[KN11b] U. Klein and K. S. Namjoshi. Formalization and Automated Verification of RESTful Behavior. Technical report, Bell Labs; Courant Institute of Mathematical Sciences, NYU TR2011-938, 2011. (Cited on page 115.)

[KP11] U. Klein and A. Pnueli. Revisiting Synthesis of GR(1) Specifications. In *Hardware and Software: Verification and Testing (Proceedings of HVC'10)*, volume 6504 of *Lect. Notes in Comp. Sci.*, pages 161–181. Springer-Verlag, 2011. (Cited on page 10.)

[KPP05] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Comp.*, 200(1):36–61, July 2005. (Cited on page 7.)

[KPP09] H. Kugler, C. Plock, and A. Pnueli. Controller synthesis from lsc requirements. In *Proc. Fundamental Approaches to Software Engineering (FASE'09), volume 5503 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 79–93, 2009. (Cited on pages 8 and 41.)

[KS09] H. Kugler and I. Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Proc.* 15th *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), volume 5505 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 77–91, 2009. (Cited on pages 8 and 41.)

[KV05] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. 46th IEEE Symp. Found. of Comp. Sci.*, 2005. (Cited on page 41.)

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, volume 193 of *Lect. Notes in Comp. Sci.*, pages 196–218. Springer-Verlag, 1985. (Cited on page 122.)

[Ltd99] ARM Ltd. AMBA specification (rev. 2). Available from www.arm.com, 1999. (Cited on page 8.)

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Clifs, 1989. (Cited on pages 118 and 142.)

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984. (Cited on page 5.)

[Nam07] K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007. (Cited on page 150.)

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994. (Cited on page 145.)

[PK09] A. Pnueli and U. Klein. Synthesis of programs from temporal property specifications. In *Proc. 7th ACM/IEEE Intl. Conference on Formal Methods and Models for Codesign*, pages 1–7. IEEE Press, 2009. (Cited on pages 8 and 44.)

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Found. of Comp. Sci.*, pages 46–57, 1977. (Cited on pages 1 and 122.)

[Pnu06] A. Pnueli. Verification and synthesis of reactive programs. Marktoberdorf Summer School Lectures, August 2006. (Cited on page 18.)

[PP06] N. Piterman and A. Pnueli. Faster solution of Rabin and Streett games. In *Proc. 21st Symposium on Logic in Computer Science*. IEEE, IEEE press, 2006. (Cited on pages 41, 52, 63, 64, and 153.)

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006. (Cited on pages 3,

7, 8, 14, 16, 17, 18, 26, 32, 35, 36, 38, 39, 41, 42, 43, 44, 52, 64, 79, 80, 82, 83, 85, 91, 92, 95, 101, 102, 105, 107, 153, and 156.)

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989. (Cited on pages 5, 6, 14, 40, 43, 50, 62, 63, 84, 111, and 157.)

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 652–671. Springer-Verlag, 1989. (Cited on pages 42, 52, 53, 63, 89, 153, and 157.)

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc.* 7th *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 82–97, 2001. (Cited on page 150.)

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *R. Alur and T. Henzinger, editors,* Proc. 8th Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of* Lect. Notes in Comp. Sci.*, Springer-Verlag*, pages 184–195, 1996. (Cited on pages 87, 105, and 106.)

[PSZ10] A. Pnueli, Y. Sa'ar, and L. D. Zuck. JTLV: A framework for developing verification algorithms. pages 171–174, 2010. (Cited on page 87.)

[PZ08] A. Pnueli and A. Zaks. On the merits of temporal testers. In *25 Years of Model Checking*, volume 5000 of *Lect. Notes in Comp. Sci.*,

pages 172–195. Springer-Verlag, 2008. (Cited on pages 15, 28, 52, 90, and 103.)

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in *cesar*. In M. Dezani-Ciancaglini and M. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lect. Notes in Comp. Sci.*, pages 337–351. Springer-Verlag, 1982. (Cited on pages 115, 144, and 158.)

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972. (Cited on pages 5 and 40.)

[RBTJ06] M. Roveri, R. Bloem, A. Tschaltev, and B. Jobstmann. Personal Communication, 2006. (Cited on pages 8 and 17.)

[SS09] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for ltl games. In *Proc.* 9th *Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'09), Austin, Texas*, pages 77–84. IEEE press, 2009. (Cited on page 8.)

[SSR08] S. Sohail, F. Somenzi, and K. Ravi. A hybrid algorithm for LTL games. In *Proc. of the* $9^t h$ *conference on Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lect. Notes in Comp. Sci.*, pages 309–323. Springer-Verlag, 2008. (Cited on page 8.)

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with application to temporal logic. *Theor. Comp. Sci.*, 49:217–237, 1987. (Cited on page 44.)

[VHB+03] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. JPF web page: `http://babelfish.arc.nasa.gov/trac/jpf`. (Cited on page 155.)

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986. (Cited on page 144.)

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. and Cont.*, 115(1):1–37, 1994. (Cited on pages 89 and 90.)

[W3C05] Uniform Resource Identifier (URI): Generic Syntax. W3C RFC 3986, 2005. (Cited on pages 116 and 127.)

[W3C07] SOAP version 1.2 part 1: Messaging framework (second edition). W3C Recommendation, 2007. `http://www.w3.org/TR/soap12-part1/`. (Cited on page 114.)

[Wil01] T. Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Soc. Math. Belg.*, 8(2), May 2001. (Cited on page 16.)

[WTM09] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon temporal logic planning for dynamical systems. In *IEEE Conference on Decision and Control*, pages 5997–6004. IEEE press, 2009. (Cited on pages 8 and 41.)

[WTM10a] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Automatic synthesis of robust embedded control software. In *AAAI Spring Sym-*

posium on Embedded Reasoning: Intelligence in Embedded Systems, 2010. (Cited on pages 8 and 41.)

[WTM10b] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Receding horizon control for temporal logic specifications. In *Hybrid Systems: Computation and Control*, Lect. Notes in Comp. Sci. Springer-Verlag, 2010. (Cited on pages 8 and 41.)