FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Property-Based Logic Synthesis for Rapid Design Prototyping (Deliverable 2.2/1)

Due date of deliverable: September 1, 2005
Actual submission date: September 1, 2005

Start date of project: January 1, 2004 Duration: Three years

Organisation name of lead contractor for this deliverable: TU Graz

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Roderick Bloem rbloem@ist.tugraz.at.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 2.2/1 described in contract number 507219.

The information in this document is provided ”as is”, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	July 15, 2005	Creation	Jobstmann	All
0.2	July 18, 2005	Include nondeterministic approach	Jobstmann	All
0.3	July 21, 2005	Extend nondeterministic approach	Jobstmann	2
0.4	August 1,2005	Include Reactivity[1] Synthesis	Pnueli, Jobstmann	All
0.5	August 2, 2005	Introduction, Summarize complete approach	Bloem, Jobstmann	1,4
0.6	August 3, 2005	Summary, conclusions	Bloem, Jobstmann	Front matter, 5
0.7	August 4, 2005	Glossary and minor revisions	Bloem, Jobstmann	All
0.8	August 5, 2005	Revision	Pnueli	All
0.9	August 8, 2005	Minor changes and release for internal review	Bloem	1,4,5
1.0	August 25, 2005	Address comments of internal reviewer Ziv Nevo	Jobstmann	2,4
1.1	August 31, 2005	Release public version	Jobstmann	

Authors

Roderick Bloem
Barbara Jobstmann
Amir Pnueli

Executive Summary

This document describes the theory of property-based synthesis. Property-based synthesis deals with automatically synthesizing a design from a specification given in the linear-time fragment of PSL. We describe three different approaches to handle property-based synthesis. These approaches are complementary: they are able to synthesize different subsets of PSL with different degrees of computational complexity. The more complete the fragment is, the more expensive is synthesis, and we show three different tradeoffs to this question.

The *nondeterministic approach* is based on an automata oriented approach to PSL model checking. It can only synthesize a subset of PSL, but can do so very efficiently. This approach is likely to be too limited for general synthesis but is very well suited for a limited synthesis problem, namely that of completion of partially implemented systems. Completion of partially implemented systems can be used

to repair faulty systems, a novel and very relevant application, at a reasonable cost [SB05].

The approach of synthesis from *generalized reactivity*[1] specifications works for a larger class of specifications, more precisely for all specifications expressible using a generalized reactivity[1] acceptance condition. The basic idea of this approach is to transform the specification into a particular form to which a triply nested fixpoint algorithm can be applied. Its complexity is expected to be feasible even for large sets of specifications.

Third, we present a *complete approach* that is applicable to the entire linear-time fragment of PSL. Its complexity bound is doubly exponential. This approach is primarily applicable for cases in which the other two approaches fail.

Purpose

The purpose of this document is to describe the research done on property based synthesis in the PROSYD project. The document provides an overview of different methods to synthesize a system from a set of properties.

Intended Audience

This guide is intended for researchers working on PSL or a similar specification language, who are interested in automatic synthesis. It is assumed that readers are familiar with the notions and terms related to PSL, have a good understanding of model checking, of game theory, and of automata theory, including tree automata and alternating automata on infinite words.

Background

Synthesis of linear-time formulas is closely related to Church's problem of synthesis for S1S [Chu63]. It was formalized by Pnueli and Rosner [PR89a]. Pnueli and Rosner's approach requires determinization of the automata, which is very expensive and intricate [Saf88]. This step is avoided in all methods presented here.

Contents

Table of Figures

Glossary

Acceptance Condition
A condition defining how an infinite automaton accepts an input object. We use Büchi and co-Büchi acceptance conditions both defined by a set of states F. An input word is Büchi accepted by an automaton, if the set of states visited infinitely often while reading the input word intersects the set F. Dually, a word is co-Büchi accepted if the set of states visited infinitely often does not intersect F.

Alternating Tree Automaton
An automaton with an arbitrary branching mode running on trees.

Atomic Proposition
An atomic proposition of a formula in a propositional logic corresponds to signals in a design or implementation.

AWT
Alternating Weak Tree Automaton. An alternating tree automaton with a particularly structured state space. The states are partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that in each transition, the automaton either stays at the same set or moves to a set smaller in the partial order.

BDD
Binary Decision Diagram. A data structure for storing sets upon which many model checkers are based.

Branching Mode
The branching mode is a way to classify automata. We distinguish between four branching modes: Deterministic, nondeterministic, universal, and alternating. In a deterministic automaton, the transition function maps from state and letter to a single state. The transition functions of nondeterministic and universal automata map to sets of states. The automata differ in the way they accept an input word or tree. In a nondeterministic automaton the suffix of the word or tree should be accepted by one of the states in the set. In the universal automaton all states in the set have to accept the suffix. An alternating automaton can have nondeterministic and universal edges.

GR[1] Formula
A generalized Reactivity[1] formula. See below.

Infinite Game
A finite state machine on which two players, the protagonist and the antagonist, determine the run, by each determining part of the input. The game comes with a winning condition and the task of the protagonist is to make sure that the run satisfies this condition.

Invariant
An invariant is a special case of a safety property. It sets a constraint on the state of the program that must hold in all states. For instance, "`a` and `b` are never 1 simultaneously."

Language Emptiness
The language of an automaton is empty iff the automaton accepts no input object (word or tree), that means there is no accepting run for this automaton.

LTL
Linear Temporal Logic or Linear-time temporal logic. LTL is a temporal logic for property specification in formal verification [Pnu77].

LTL Game
An infinite game where the winning condition is given as LTL formula. All plays in which the sequence of states visited fulfill the given formula are winning for the protagonist. Otherwise the antagonist wins.

Mu Calculus
A calculus of predicates and binary relations which enables writing and solving relational equations among states.

NBT
Nondeterministic Büchi Tree Automaton. An alternating tree automaton with Büchi acceptance condition and nondeterministic branching mode.

NBW
Nondeterministic Büchi Word Automaton. An alternating automaton with Büchi acceptance condition and nondeterministic branching mode. The automaton runs on words.

PSL
Property Specification Language, the language for specification of designs upon which PROSYD is based.

PSL Game
Similar to an LTL game but with a PSL formula as winning condition.

Reactivity Formula
A PSL formula of the form $\bigwedge_i^k (\texttt{inf\_oft}\ p_i \implies \texttt{inf\_oft}\ q_i)$, where "$\texttt{inf\_oft}\ p$" is an abbreviation for "$\texttt{always eventually!}\ p$", and p, q are Boolean formulas.

Reactivity[1] Formula
A reactivity formula with $k = 1$.

Reactivity[1] Formula – Generalized
A PSL formula of the form

$$(\texttt{inf\_oft}\ p_1 \wedge \cdots \wedge \texttt{inf\_oft}\ p_m) \implies (\texttt{inf\_oft}\ q_1 \wedge \cdots \wedge \texttt{inf\_oft}\ q_n),$$

where $p_1, \ldots, p_m, q_1, \ldots, q_n$ are Boolean formulas.

Realizable
A given formula ψ over a sets of input I and output O signal is realizable if there exists a strategy $f : (2^I)^* \rightarrow 2^O$ such that all the computations of the system generated by f satisfy ψ. Intuitively, a specification is realizable if there exists a system that can respond in such a way that independent of the input values the environment chooses the combination of inputs and outputs always fulfills the given formula.

Safety Property
A safety property states that something bad should not happen. For instance, "a is never 1 in two consecutive clock ticks."

Street Automaton
An ω-automaton in which the acceptance condition is presented by a set of acceptance pairs. Each acceptance pair consists of two sets of the automaton states. An infinite run r is accepted by a pair (S, T) if its infinity set (states which are visited

infinitely often by r) is either disjoint from S or has a nonempty intersection with T. A run is accepted by the automaton if it is accepted by all pairs.

Street[1] Automaton
A Street automaton with a single pair.

Street Automaton – Generalized
An ω-automaton in which the acceptance condition is presented by a set of acceptance pairs. Each acceptance pair consists of two families of sets of automaton states. An infinite run r is accepted by a pair $(\{S_1,\ldots,S_m\},\{T_1,\ldots,T_n\})$ if the infinity set of r is either disjoint from some S_i or has a nonempty intersection with every T_j, $j = 1,\ldots,n$. A run is accepted by the automaton if it is accepted by all pairs of the automaton.

Street[1] Automaton – Generalized
A generalized Street automaton with a single pair.

Synthesis
The process of automatically generating a design from a given specification. Formally, check if the given specification is realizable and find a witness.

UCT
Universal co-Büchi Word Automaton. An alternating tree automaton with co-Büchi acceptance condition and universal branching mode.

Winning Strategy
A recipe with which a player is guaranteed to win an infinite game, no matter what the other player does. A finite state strategy may depend on a finite memory of the past, i.e., the move the strategy suggests can depend on previous moves of the two players. A memoryless strategy depends only on the current state of the game.

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 prevalent approach to solving the synthesis problem was by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person 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 programs from temporal specification which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. 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 [Saf88]. This double translation may cause a complexity that is doubly exponential in the size of φ. Once the Rabin automaton is obtained, the game can be solved in time n^k, where n is the number of states of the automaton and k is the number of accepting pairs. In our work on synthesis and realizability, we restrict our attention to the linear fragment of PSL. As shown in [KMTV00], checking realizability of branching time logics is very complex and may reach, in the worst case, triply exponential complexity.

The high complexity established in [PR89a] and the intricacy of Safra's determinization construction have caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to implement it.

There are two basic ways out of this dilemma. The first is to limit the expressiveness of the temporal logic, with the benefit of a better computational complexity. The second option is to circumvent Safra's construction, which retains the disadvantage of a higher complexity, but has the advantage that it is applicable to the full extent of logics such as LTL or PSL.

If the specification of the design to be synthesized is restricted to simpler automata or partial fragments of PSL, 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 to games (and hence synthesis problems) where the acceptance condition is one of the formulas `always` p, `eventually!` q, `always eventually!` p, or `eventually! always` q. A more recent paper is [AT04] which presents efficient synthesis approaches for the fragment consisting of a Boolean combinations of formulas of the form `always` p.

Chapter 2 and 3 proceed by restricting the logic that is allowed for synthesis. Chapter 2 presents a synthesis procedure that builds on nondeterministic Büchi automata. This approach is very efficient, but handles a relatively modest subset of PSL. We show an application of this approach to complete the details of partially implemented systems. Partially implemented systems turn up in the repair of faulty systems, which need to be modified to fulfill their specification [SB05].

Chapter 3 can be viewed as a generalization of the results of [AMPS98] and [AT04] into the wider class of *Generalized Reactivity*[1] formulas (GR[1]). Following the developments in [KPP05], we show how any synthesis problem whose specification is a GR[1] formula can be solved in time N^3. Furthermore, we present a symbolic algorithm for extracting a design (program) which implements the specification. We make an argument that the class of GR[1] formulas is sufficiently expressive to provide complete specifications of many designs.

Chapter 4 presents a complete approach to synthesis of PSL. It is based on an approach by [KV05] to avoid Safra's construction. Even though the complexity of the procedure remains doubly exponential in the size of the specification, the implementation is simpler and likely to be more efficient.

The three approaches complement each other by occupying different points on the tradeoff between complexity and expressiveness. None of the approaches by itself will yield a solution that is generally applicable. Rather, the correct approach should be selected depending on the properties provided. Ideally, one could try each approach in turn, from cheap to expensive, and utilize its results if the approach is able to synthesize a system.

2 Nondeterministic Approach

The approach described in this section uses nondeterministic automata to represent the given specification. It is less expensive than the other approaches and works only for a subset of PSL specifications. It is mainly used to repair faulty systems based on completing partially implemented systems. To repair a faulty system we "free" components likely to be responsible for the fault and get a partially implemented system. Then we search for a new implementation of the free components. Therefore, we explain this approach starting from a partially implemented system. At the end of this chapter we show how the approach can be used to perform synthesis without having parts of the system already implemented.

Starting from a partially implemented system we construct a game between two players, the environment and the system. The environment can determine the inputs values of the game. The system tries to choose an implementation for the unimplemented parts in such a way that the given specification is fulfilled. The specification defines the winning condition of this two-player game. A winning strategy for the system corresponds to a valid implementation of the unrestricted parts of the original system. Depending on the winning condition we obtain either a finite state or a memoryless winning strategy. In 2.2, we show how to construct the game out of a partially implemented system and we define a winning condition. In Section 2.3, we describe a way to solve this game and we explain how to obtain a winning strategy. A finite state strategy for the game corresponds to an implementation that adds states to the system. Since in the repair setting we want an implementation that is as close as possible to the original system, we need a memoryless strategy. In 2.4, we show that it is NP-complete to decide whether a memoryless strategy exists, and in 2.5, we present a heuristic to construct a memoryless strategy. We complete the approach by extracting an implementation from a memoryless strategy. In the last two sections we present an example for the repair application and we describe complete synthesis based on this approach.

2.1 Preliminaries

In this section, we describe the necessary theoretical background for this approach. We assume basic knowledge of the μ-calculus, PSL, and the translation of PSL to Büchi automata [BDBF$^+$05].

A *game* G over AP is a tuple $(S, s_0, I, C, \delta, \lambda, F)$, where S is a finite set of states, $s_0 \in S$ is the initial state, I and C are finite sets of environment inputs and system choices, $\delta : S \times I \times C \rightharpoonup S$ is the partial transition function, $\lambda : S \to 2^{AP}$ is the labeling function, and $F \subseteq S^\omega$ is the winning condition, a set of infinite sequences of states. With the exception of this section and Section 2.5, we will assume that δ is a complete function. Intuitively, a game is a partially implemented finite state machine together with a specification. The environment inputs are as usual, and the system choices C represent the freedom of implementation. The challenge is to find proper values for C such that F is satisfied.

Given a game $G = (S, s_0, I, C, \delta, \lambda, F)$, a *(finite state) strategy* is a tuple $\sigma = (Q, q_0, \mu)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\mu : Q \times S \times I \to 2^{C \times Q}$ is the *move function*. Intuitively, a strategy automaton fixes a set of possible responses to an environment input, and its response may depend on a finite memory of the past. Note that strategies are nondeterministic. We need nondeterminism in the following in order to have maximal freedom when we attempt to convert a finite state strategy to a memoryless strategy. For the strategy to be winning, a winning play has to ensue for any nondeterministic choices of the strategy.

A *play* on G according to σ is a finite or infinite sequence $\pi = q_0 s_0 \xrightarrow{i_0 c_0} q_1 s_1 \xrightarrow{i_1 c_1} \dots$ such that $(c_i, q_{i+1}) \in \mu(q_i, s_i, i_i)$, $s_{i+1} = \delta(s_i, i_i, c_i)$, and either the play is infinite, or there is an n such that $\mu(q_n, s_n, i_n) = \emptyset$ or $\delta(s_n, i_n, c_n)$ is not defined, which means that the play is finite. A play is *winning* if it is infinite and $s_0 s_1 \cdots \in F$. (If $\mu(q_n, s_n, i_n) = \emptyset$, the strategy does not suggest a proper system choice and the game is lost.) A strategy σ is *winning* on G if all plays according to σ on G are winning.

A *memoryless strategy* is a finite state strategy with only one state. We will write a memoryless strategy as a function $\sigma : S \times I \to 2^C$ and a play of a memoryless strategy as a sequence $s_0 \xrightarrow{i_0 c_0} s_1 \xrightarrow{i_1 c_1} \dots$, leaving out the state of strategy automaton.

We extend the labeling function λ to plays: the *output word* is $\lambda(\pi) = \lambda(s_0)\lambda(s_1)\dots$. Likewise, the *input word* is $\iota(\pi) = i_0 i_1 \dots$, the sequence of system inputs. The *output language (input language)* $L(G)$ $(I(G))$ of a game is the set of all $\lambda(\pi)$ $(\iota(\pi))$ with π winning.

A *safety game* has the condition $F = \{s_0 s_1 \cdots \mid \forall i : s_i \in A\}$ for some $A \subseteq S$. The winning condition of an *LTL or PSL game* is the set of sequences satisfying an LTL or PSL formula φ. In this case, we will write φ for F. We restrict ourselves to linear-time segment of PSL. LTL and PSL only differ in the way of constructing an automaton for a formula, see [BDBF$^+$05] for more details. *Büchi games* are defined by a set $B \subseteq Q$, and require that a play visit the Büchi constraint B infinitely often. For such games, we will write B for F.

Following the construction proposed in [BDBF$^+$05] we can convert a PSL formula φ over the set of atomic propositions AP to a Büchi game $A = (Q, q_0, 2^{AP}, C, \delta, \lambda, B)$ such that $I(A)$ is the set of words satisfying φ. The system choice models the nondeterminism of the automaton.

In order to solve games, we introduce some notation. For a set $A \subseteq S$, the set $\mathsf{MX}A = \{s \mid \forall i \in I \, \exists c \in C s' \in A : (s, i, c, s') \in \delta\}$ is the set of states from which the system can force a visit to a state in A in one step. The set $\mathsf{M}A\mathsf{U}B$ is defined by the μ-calculus formula $\mu Y. B \cup \mathsf{MX}(A \cap Y)$. It defines the set of states from which the system can force a visit to B without leaving A. The *iterates* of this computation are $Y_0 = B$ and $Y_{j+1} = Y_j \cup (A \cap \mathsf{MX} Y_{j-1})$ for $j > 0$. From Y_j the system can force a visit to B in at most j steps. Note that there are only finitely many distinct iterates.

We define $\mathsf{MG}A = \nu Z.A \cap \mathsf{MX}Z$, the set of states from which the system can avoid leaving A. Since the system has to stay in a safe region in order to win a safety game, we use $\mathsf{MG}A$ to compute the winning region of the system in safety games. For a Büchi game, we define $W = \nu Z.\mathsf{MX}\,\mathsf{M}Z\mathsf{U}(Z \cap B)$. The set W is the set of states from which the system can win the Büchi game. Note that these fixpoints are similar to the ones used in model checking of fair CTL and are easily implemented symbolically.

Using these characterizations, we can compute memoryless strategies for safety and Büchi games [Tho95]. For a safety game with condition A, the strategy $\sigma(s,i) = \{c \in C \mid \exists s' \in \mathsf{MG}A : (s,i,c,s') \in \delta\}$ is winning if and only if $s_0 \in \mathsf{MG}A$. For a Büchi game, let $W = \nu Z.\mathsf{MX}\,\mathsf{M}Z\mathsf{U}(Z \cap B)$. Let Y_1 through Y_n be the set of distinct iterates of $\mathsf{M}W\mathsf{U}(W \cap B) = W$. We define the *attractor strategy* for B to be

$$\begin{array}{rcl} \sigma(s,i) & = & \{c \in C \mid \exists j,k < j, s' \in Y_k : s \in Y_j \setminus Y_{j-1}, (s,i,c,s') \in \delta\} \cup \\ & & \{c \in C \mid s \in Y_0, \exists s' \in W, \exists i \in I : (s,i,c,s') \in \delta\}. \end{array}$$

The attractor strategy brings the system ever closer to B, and then brings it back to a state from which it can force another visit to B.

2.2 Constructing a Game

Suppose that we are given a partially implemented system and we want it to fulfill a given PSL specification. Our problem is to find a simple extension to the system such that the specification holds on the complete system.

Using a small example we explain how the approach works. Assume we are given the circuit depict in Figure 1. It represents a simple arbiter with a missing implementation of one of its components. The arbiter should adhere to the following specification $\varphi = \texttt{always}\ (r \rightarrow (a \vee \texttt{next!}\ a)) \wedge \texttt{always}\ (a \rightarrow \neg\texttt{next!}\ a) \wedge \texttt{always}\ (\neg r\ \texttt{until}\ a)$.

The unimplemented component is labeled with a question mark and we aim for an output function f of this component in terms of the input and the state variables. The game between the system and the environment proceeds as follows: In every time step the environment determines the input signals. In our example we have a single input signal named r. Once the environment has selected the input values, the system chooses an output value for the unimplemented component. Figure 2 shows the corresponding game graph. Since the given specification is a PSL formula we obtain a PSL game. In Section 2.3 we show a way of solving these games.

If the given specification is an invariant, e.g., $\texttt{always}\ (\mathrm{D1} = \mathrm{D0})$, we mark the states violating the invariant as *bad states* and obtain a Safety game. We can solve the game using the algorithm sketched in the preliminary section. Safety games always have a memoryless winning strategy for the winning player. In Section 2.6 we show how to extract an implementation from a memoryless strategy of the system.

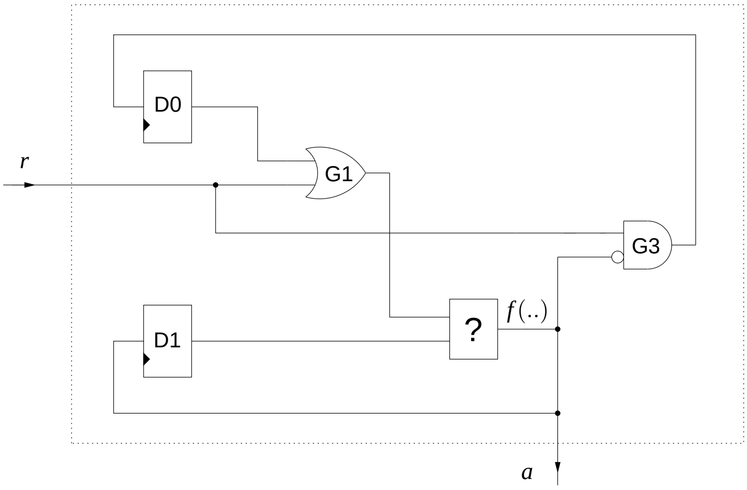

Figure 1: Partially implemented arbiter circuit

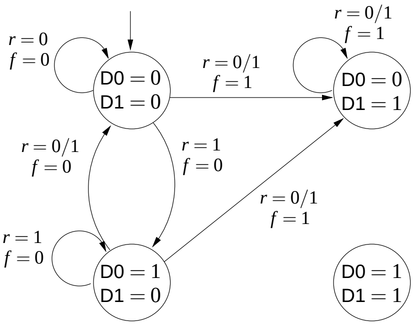

Figure 2: Game graph corresponding to the partially implemented system in Figure 1. The circles of the graph are labeled with state variables and the edges are labeled with input and system choice variables (r, f).

2.3 Solving Games

In general, deciding LTL games is 2EXPTIME-complete [PR89b] and the problem is even harder for PSL because of the succinctness of PSL. The common approach to solve these games also includes a highly complex determinization construction [Saf88]. Therefore we propose a less complicated and less expensive approach applicable for a subset of PSL games. We discuss the restrictions of this approach at the end of this section. In the following we describe the approach formally and prove its applicability to PSL games.

Given two games $G = (S, s_0, I_G, C_G, \delta_G, \lambda_G, F_G)$ and $A = (Q, q_0, 2^{AP}, C_A, \delta_A, \lambda_A, F_A)$, let the *product game* be $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, F)$, where $\delta((s,q), i_G, (c_G, c_A)) = \big(\delta_G(s, i_G, c_G), \delta_A(q, \lambda_G(s), c_A)\big)$, $\lambda(s,q) = \lambda_G(s)$, and $F = \{(s_0, q_0), (s_1, q_1), \cdots \mid s_0, s_1, \cdots \in F_G$ and $q_0, q_1, \cdots \in F_A\}$. Intuitively, the output of G is fed to the input of A, and the winning conditions are conjoined. Therefore, the output language of the product is the intersection of the output language of the first game and the input language of the second.

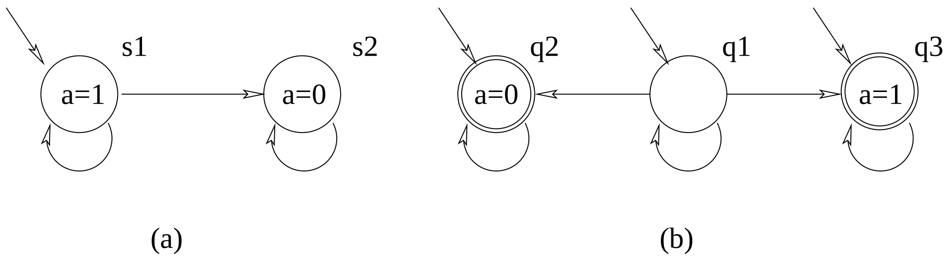

Figure 3: (a) Game in which the environment can assign the value for variable a. (b) Automaton for `eventually! always` $(a = 1) \vee$ `eventually! always` $(a = 0)$.

Lemma 1. *For games G, A,* $L(G \triangleright A) = L(G) \cap I(A)$.

Lemma 2. *Let G and A be games. If a memoryless winning strategy for* $G \triangleright A$ *exists, then there is a finite state winning strategy* σ *for G such that for all plays* π *of G according to* σ, $\lambda(\pi) \in L(G)$ *and* $\lambda(\pi) \in I(A)$.

The finite state strategy σ is the product of A and the memoryless (single state) strategy for $G \triangleright A$. If $F_G = S^\omega$, then σ is the winning strategy for the game G with the winning condition defined by A. The following result (an example of *game simulation*, cf. [Tho95]) follows from the lemma above 2.

Theorem 1. *Let* $G = (S, s_0, I, C, \delta, \lambda, \varphi)$ *be a PSL game, let* G' *be as G but with the winning condition* S^ω*, and let A be a Büchi game with* $I(A) = L(G)$*. If there is a memoryless winning strategy for the Büchi game* $G' \triangleright A$ *then there is a finite state winning strategy for G.*

Note that the converse of the theorem does not hold. In fact, Harding [Har05] shows that we are guarantee to find a winning strategy iff the game fulfills the property and the automaton is *trivially determinizable*, i.e., we can make it deterministic by removing edges without changing the language.

For example, there is no winning strategy for the game shown in Fig. 3. If the automaton for the property `eventually! always` $(a = 1) \vee$ `eventually! always` $(a = 0))$ moves to the state q_3, the environment can decide to move to s_2 (set $a = 0$), a move that the automaton cannot match. If, on the other hand, the automaton waits for the environment to move to s_2, the environment can stay in s_1 forever and thus force a non-accepting run. Hence, although the game fulfills the formula, we cannot give a strategy. Note that this problem depends not only on the structure of the automaton, but also on the structure of the game. For instance, if we remove the edge from s_1 to s_2, we can give a strategy for the product.

In general, the translation of an LTL formula to a deterministic automaton requires a doubly exponential blowup and the best known upper bound for deciding whether a translation is possible is EXPSPACE [KV98]. To prevent this blowup, we can either use heuristics to reduce the number of nondeterministic states in the automaton [ST03], or we can use a restricted subset of temporal logic. Maidl [Mai00] shows that translations in the style of [GPVW95] yield deterministic automata for the formulas in the set LTL$^{\mathrm{det}}$, which is defined as follows: If φ_1 and φ_2 are LTL$^{\mathrm{det}}$ formulas, and p is a predicate, then p, $\varphi_1 \wedge \varphi_2$, `next!` φ_1, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $(p \wedge \varphi_1)$ `until!` $(\neg p \wedge \varphi_2)$, and $(p \wedge \varphi_1)$ `until` $(\neg p \wedge \varphi_2)$ are LTL^{det} formulas. Note that this set includes invariants ($\texttt{always}\ p$) and $\neg p\ \texttt{until!}\ p = \texttt{eventually!}$ LTL$^{\text{det}}$ describes the intersection of LTL and CTL. In fact, deterministic Büchi automata describe exactly the properties expressibly in the alternation-free μ-calculus, a superset of CTL [KV98].

Alur and La Torre [AT04] define a set of formulas for which we can compute deterministic automata using a different tableau construction. These formulas are Boolean combinations of subformulas written using a restricted set of operators. They give an appropriate construction for the case in which the subformulas are constructed using only `eventually!`, `next!`, and $\wedge$. In contrast, if the subformulas are built using `eventually!`, $\vee$, and $\wedge$ or `always` and `eventually!`, they show that the size of a corresponding deterministic automaton is necessarily doubly exponential in the size of the formula. Since trivially deterministic automata can be made deterministic by removing edges, they can be no smaller than the smallest possible deterministic automaton and thus there are no exponential-size trivially deterministic automata for the latter two groups.

2.4 Memoryless Strategies are NP-Complete

The finite state strategy corresponding to the product game defined in the last section may be quite awkward to implement as it requires the program to keep track of the state of the automaton. This means that we have to add extra state, which we have to update whenever a variable changes that is referenced in the specification. Instead, we wish to construct a memoryless strategy. Such a strategy corresponds to an implementation of the component that does not require additional state.

It follows from the results of Fortune, Hopcroft, and Wyllie [FHW80] that given a directed graph G and two nodes v and w, it is NP-complete to compute whether there are node-disjoint paths from v to w and back. Assume that we build a game G' based on the graph G. The acceptance condition is that v and w are visited infinitely often, which can easily be expressed by a Büchi automaton. Since the existence of a memoryless strategy for G' implies the existence of two node-disjoint paths from v to w and back, we can deduce the following theorem.

Theorem 2. *Deciding whether a game with a winning condition defined by a Büchi automaton has a memoryless winning strategy is NP-complete.*

It follows that for PSL games there is no algorithm to decide whether there is a memoryless winning strategy that runs in time polynomial in the size of the underlying graph, unless P = NP, even if a finite state strategy is given.

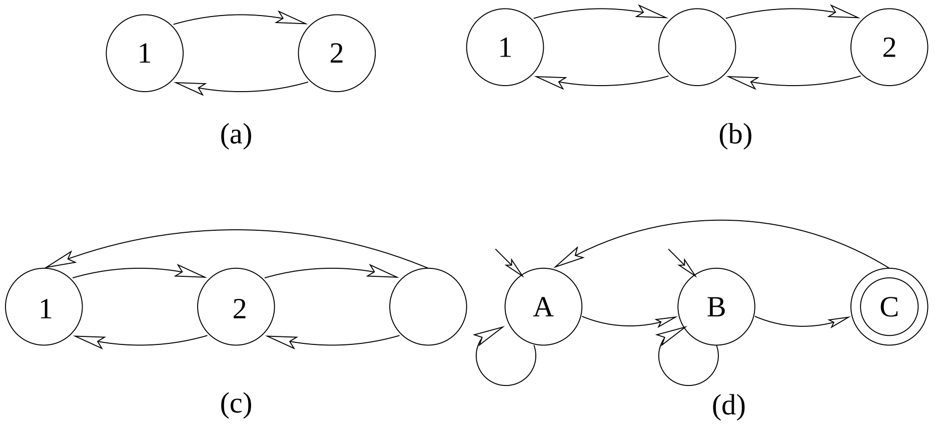

Figure 4: Fig. a, b, c show three games with the winning condition that states 1 and 2 are both visited infinitely often. Multiple outgoing arcs from a state model a system choice. The winning condition is defined by the Büchi automaton shown in Fig. d. For Fig. a, the strategies for States A, B, and C coincide, and a memoryless strategy exists. For Fig. b, no memoryless strategy exists, and for Fig. c, a memoryless strategy exists, but it is not equal to the intersection of all the strategies for states A, B, and C. (The strategies are contradictory for the state on the right.)

2.5 Heuristics for Memoryless Strategies

Since we cannot compute a memoryless strategy in polynomial time, we use a heuristic. Given a memoryless strategy for the product game, we construct a strategy that is common to all states of the automaton, which is our candidate for a memoryless strategy on the program game. Then, we compute whether the candidate is a winning strategy, which is not necessarily the case. Note that invariants have an automaton consisting of one state and thus the memoryless strategy for the product game is a memoryless strategy for the program game.

Recall that the product game is $G \triangleright A = (S \times Q, (s_0, q_0), I_G, C_G \times C_A, \delta, \lambda, B)$. Let $\sigma : (S \times Q) \times I_G \to 2^{C_G \times C_A}$ be the attractor strategy for condition B. Note that the strategy is immaterial on nodes that are either not reachable (under any choice of the system) or not winning (and thus will be avoided by the system). Let R be the set of reachable states of the product game, and let W be the set of winning states. We define

$$\tau'(s, i_G) = \big\{ c_G \bigm| \forall q \in Q : \big((s,q) \notin R \cap W \text{ or } \exists c_A \in C_A : (c_G, c_A) \in \sigma((s,q), i_G)\big)\big\}.$$

Intuitively, we obtain τ' by taking the moves common to all reachable, winning states of the strategy automaton.[1] See Fig. 4 for an illustration of the limitations of the approach.

If τ' is winning, then so is σ, but the converse does not hold. To check whether τ' is winning, we construct a game G' from G by restricting the transition relation to adhere to τ': $\delta' = \{(s, i, c, s') \in \delta \mid c \in \tau'(s, i)\}$. This may introduce states without a successor. We see whether we can avoid such states by computing $W' = \mathsf{MG}S$.

[1] We may treat multiple B¨uchi constraints, if present, in the same manner. This is equivalent to using the counting construction.

If we find that $s_0 \notin W'$, we cannot avoid visiting a dead-end state, and we give up trying to find an implementation. If, on the other hand, $s_0 \in W'$, we get our final memoryless strategy τ by restricting τ' to W', which ensures that a play that starts in W' remains there and never visits a dead-end. We thus reach our main conclusion in the following theorem.

Theorem 3. *If $s_0 \in W'$ then τ is a winning strategy of G.*

2.6 Extracting an Implementation

This section shows a symbolic method to extract an implementation statement from a memoryless strategy. We determinize the strategy by finding proper assignments to the system choices that can be used in the suspect locations. For any given state of the program, the given strategy may allow for multiple assignments, which gives us room for optimization.

We may not want the implementation to depend on certain variables of the program, for example, because they are out of the scope of the component that is being implemented. In that case, we can universally quantify these variables from the strategy and its winning region and check that the strategy still supplies a valid response for all combinations of state and input.

For each assignment to the system choice variables, we calculate a set $P_j \subseteq S \times I$ for which the assignment is a part of the given strategy. We can use these sets P_j to suggest the implementation "`if P0 then assign0 else if P1 then ...`", in which `Pj` is an expression that represents the set P_j. The expression `Pj`, however, can be quite complex: even for small examples it can take over a hundred lines, which would make the suggested implementation inscrutable.

We exploit the fact that the sets P_j can overlap to construct new sets A_j that are easier to express. We have to ensure that we still cover all winning and reachable states using the sets A_j. Therefore, A_j is obtained from P_j by adding or removing states outside of a *care set*. The care set consists of all states that cannot be covered by A_j because they are not in P_j and all states that must be covered by A_j because they are neither covered by an A_k with $k < j$, nor by a P_k with $k > j$. We then replace `Pj` with an expression for A_j to get our implementation suggestion.

For simultaneous assignment to many variables, we may consider generating suggestions for each variable separately, in order to avoid enumerating the domain. For example, we could assign the variables one by one instead of simultaneously.

Extracting a simple implementation is similar to multi-level logic synthesis in the presence of satisfiability don't cares and we may be able to apply multi-level minimization techniques [HS96]; the problem of finding the smallest expression for a given relation is NP-hard by reduction from 3SAT. One optimization we may attempt is to vary the order of the `Aj`s, but in our experience, the suggested implementations are typically quite readable.

```
  int got_lock = 0;
  do{
1   if (*) {
2     lock();
3     got_lock++; }
4   if (got_lock != 0) {
5     unlock();}
6   got_lock--;
7 } while(*)

  void lock() {
11  assert(L = 0);
12  L = 1; }

  void unlock(){
21  assert(L = 1);
22  L = 0; }
```

Figure 5: Locking Example

2.7 Repair Example

Since this approach is mainly used to repair faulty systems, we present an example of an application in repair.

We repair the abstract program shown in Fig. 5 [HJMS02, GV03]. This programs abstracts a class of concrete programs with different if and while conditions, all of which perform simple lock request/release operations. The method `lock()` checks that the lock is available and requests it. Vice versa, `unlock()` checks that the lock is held and releases it. The `if(*)` in the first line causes the lock to be requested nondeterministically, and the `while(*)` causes the loop to be executed an arbitrary number of times. The variable `got_lock` is used to keep track of the status of the lock (Lines 4 and 5). The assertions in Lines 11 and 21 constitute a safety property that is violated, e.g., if the loop is executed twice without requesting the lock. The fault is that the statement `got_lock--` should be placed within the scope of the preceding `if`.

Model-based diagnosis can be used to find a candidate for the repair [MSW00]. A diagnosis of the given example was performed in [CKW05] and localizes the fault in Lines 1, 6, or 7. We reject the possibility of changing Line 1 or 7 because we want the repair to work regardless of the if and while conditions in the concrete program. Instead, we look for a faulty assignment to `got_lock`. Thus, we free the RHS in Lines 3 and 6. The algorithm suggests a correct repair, `got_lock=1` for Line 3 and `got_lock=0` for Line 6. Note that we repair the program using a different fault model than the one which caused it, i.e., after the repair the program is correct, even though we did not suggest to move `got_lock--` inside the scope of the `if`.

2.8 Synthesis

We can use the ideas presented in this chapter to perform synthesis without the benefit of a partially implemented system. Starting from a specification given as (linear-time) PSL formula we construct a nondeterministic Büchi automaton. The atomic propositions representing input signals are controlled by the environment. The nondeterminism and the output signals are again left to the system and we obtain the Büchi game. We skip the next step of our approach, where we would build the product game, since we have not got a partially implemented system. Instead, we directly computed the winning strategy for the system, if one exists. Due to the fact that we have Büchi games the winning strategies is memoryless. Once we have a winning memoryless strategy we restrict the game to this strategy and obtain a system that responds correctly no matter what input values the environment chooses. Since we have nondeterministic strategies the system can still have some nondeterministic choice. We remove this choice by adding an additional parameter to the system, which allows us to switch between different correct behaviors of the system. This is advantageous if the system is used as a test driver. Note that this approach is also subject to the restriction mentioned in the previous sections.

3 Reactivity[1] Synthesis

In this chapter we present an approach to the synthesis of designs whose specification is given by a generalized *reactivity*[1] formulas. That is, formulas of the form

$$\begin{aligned}(\texttt{always}\ \texttt{eventually!}\ p_1 \wedge \cdots \wedge \texttt{always}\ \texttt{eventually!}\ p_m) \quad \Longrightarrow & \qquad (1)\\ (\texttt{always}\ \texttt{eventually!}\ q_1 \wedge \cdots \wedge \texttt{always}\ \texttt{eventually!}\ q_n,) &\end{aligned}$$

where $p_1,\ldots,p_m,q_1,\ldots,q_n$ are *assertions* (state formulas). We refer to this class of properties as GR[1] properties.

3.1 Preliminaries

Linear Temporal Logic

In this section we restrict our attention to the linear segment of PSL. From this segment we single out the part that can be expressed by the operators `next!`, `until`, `eventually!`, and `always`. Since this fragment is isomorphic to the linear temporal logic LTL, we will often refer to formulas in this fragment as LTL-formulas.

We are interested in the question of realizability of LTL specifications. Assume two sets of variables $\mathcal{X}$ and $\mathcal{Y}$. Intuitively $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of system variables. With no loss of generality, we assume that all variables are Boolean. Obviously, the more general case that $\mathcal{X}$ and $\mathcal{Y}$ range over arbitrary finite domains can be reduced to the Boolean case. *Realizability* amounts to checking whether there exists an *open controller* that satisfies the specification. Such a controller can be represented as an automaton which, at any step, inputs values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables. Below we formalize the notion of checking realizability and *synthesis*, namely, the construction of such controllers.

We are interested in a subset of LTL for which we solve realizability and synthesis in polynomial time. The specifications we consider are of the form $\varphi = \varphi_e \rightarrow \varphi_s$. We require that φ_α for $\alpha \in \{e,s\}$ can be rewritten as a conjunction of the following parts.

- φ_i^α - a Boolean formula which characterizes the initial states of the implementation.

- φ_t^α - a formula of the form $\bigwedge_{i\in I}$ `always` B_i where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form `next!` v where $v \in \mathcal{X}$ if $\alpha = e$, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
- φ_g^α - a formula of the form $\bigwedge_{i\in I}$ `always eventually!` B_i where each B_i is a Boolean formula.

It turns out that most of the specifications written in practice can be rewritten to this format. In Section 3.5 we discuss also cases where the formulas φ_g^α have also subformulas of the form `always` $(p \rightarrow$ `eventually!` $q)$ where p and q are Boolean formulas, and additional cases which can be converted to the GR[1] format.

Game Structures

We reduce the realizability problem of an LTL formula to the decision of winner in games. We consider two-player games played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. Formally, we have the following.

A *game structure* (GS) $G : \langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi\rangle$ consists of the following components.

- $V = \{u_1, \ldots, u_n\}$: A finite set of typed *state variables* over finite domains. With no loss of generality, we assume they are all Boolean. We define a *state* s to be an interpretation of V, assigning to each variable $u \in V$ a value $s[u] \in \{0,1\}$. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to Boolean expressions over V in the usual way. An *assertion* is a Boolean formula over V. A state s satisfies an assertion φ denoted $s \models \varphi$, if $s[\varphi] = \mathbf{true}$. We say that s is a φ-state if $s \models \varphi$.
- $\mathcal{X} \subseteq V$ is a set of *input variables*. These are variables controlled by the environment. Let X denote the possible valuations to variables in $\mathcal{X}$.
- $\mathcal{Y} = V \setminus \mathcal{X}$ is a set of *output variables*. These are variables controlled by the system. Let Y denote the possible valuations for the variables in $\mathcal{Y}$.
- Θ is the initial condition. This is an assertion characterizing all the initial states of G. A state is called *initial* if it satisfies Θ.
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the transition relation of the environment. This is an assertion, relating a state $s \in \Sigma$ to a possible input value $\vec{x}' \in X$, by referring to unprimed copies of $\mathcal{X}$ and $\mathcal{Y}$ and primed copies of $\mathcal{X}$. The transition relation ρ_e identifies valuation $\vec{x}' \in X$ as a possible *input* in state s if $(s, \vec{x}') \models \rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ where $(s, \vec{x}')$ is the joint interpretation which interprets $u \in V$ as $s[u]$ and for $v \in \mathcal{X}$ interprets v' as $\vec{x}[v]$.
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the system. This is an assertion, relating a state $s \in \Sigma$ and an input value $\vec{x}' \in X$ to an output value $\vec{y} \in Y$, by referring to primed and unprimed copies of V. The transition relation ρ_s identifies a valuation $\vec{y}' \in Y$ as a possible *output* in state s reading input $\vec{x}'$ if $(s, \vec{x}', \vec{y}') \models \rho_s(V, V')$ where $(s, \vec{x}', \vec{y}')$ is the joint interpretation which interprets $u \in \mathcal{X}$ as $s[u]$, u' as $\vec{x}'[u]$, and similarly for $v \in \mathcal{Y}$.

- φ is the winning condition, given by an LTL formula.

For two states s and s' of G, s' is a *successor* of s if $(s,s') \models \rho_e \wedge \rho_s$. We freely switch between $(s,\vec{x}') \models \rho_e$ and $\rho_e(s,\vec{x}') = 1$ and similarly for ρ_s. A *play* σ of G is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* namely $s_0 \models \Theta$, and *consecution* namely, for each $j \geq 0$, s_{j+1} is a successor of s_j. Let G be an GS and σ be a play of G. From a state s, the environment chooses an input $\vec{x}' \in X$ such that $\rho_e(s,\vec{x}') = 1$ and the system chooses an output $\vec{y}' \in Y$ such that $\rho_s(s,\vec{x}',\vec{y}') = \rho_s(s,s') = 1$.

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

A *strategy* for the system is a partial function $f : \Sigma^+ \times X \mapsto Y$ such that if $\sigma = s_0, \ldots s_n$ then for every $\vec{x}' \in X$ such that $\rho_e(s_n,\vec{x}') = 1$ we have $\rho_s(s_n,\vec{x}', f(\sigma \cdot \vec{x}')) = 1$. Let f be a strategy for the system, and $s_0 \in \Sigma$. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \ldots, s_i, s_{i+1}[\mathcal{X}]) = s_{i+1}[\mathcal{Y}]$, where $s_{i+1}[\mathcal{X}]$ and $s_{i+1}[\mathcal{Y}]$ are the restrictions of s_{i+1} to variable sets $\mathcal{X}$ and $\mathcal{Y}$, respectively. Strategy f is *winning* for the system from state $s \in \Sigma_G$ if all s-plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_s the set of states from which there exists a winning strategy for the system. A *strategy* for player environment, *winning strategy*, and the *winning set* W_e are defined dually. A GS G is said to be *winning* for the system if all initial states are winning for the system.

Given an LTL specification $\varphi_e \rightarrow \varphi_s$ as explained above and sets of input and output variables $\mathcal{X}$ and $\mathcal{Y}$ we construct a GS as follows. Let $\varphi_\alpha = \varphi_i^\alpha \wedge \varphi_t^\alpha \wedge \varphi_g^\alpha$ for $\alpha \in \{e,s\}$. Then, for Θ we take $\varphi_i^e \wedge \varphi_i^s$. Let $\varphi_t^\alpha = \bigwedge_{i \in I} \texttt{always}\ B_i$, then $\rho_\alpha = \bigwedge_{i \in I} \tau(B_i)$, where the translation τ replaces each instance of `next!` v by v'. Finally, we set $\varphi = \varphi_g^e \rightarrow \varphi_g^s$. We *solve* the game, attempting to decide whether the game is winning for the environment or the system. If the environment is winning the specification is *unrealizable*. If the system is winning, we *synthesize* a winning strategy which is a *working implementation* for the system as explained in Section 3.3.

Compatibility with Previous Definition of Games

In Section 2.1 we presented a general definition of a game as given by a tuple $(S, s_0, I, C, \delta, \lambda, F)$. The definition of game structure as presented here is a particular syntactic instantiation of the general definition. In this instantiation, states are represented as a valuation of state variables. The environment inputs I and system actions C are represented in the syntactic game structures as the interpretations of the input variables $\mathcal{X}$ and output variables $\mathcal{Y}$, respectively. Similarly, the transition function δ is captured by the combination of the transition relation ρ_e and ρ_s.

Fair Discrete Systems

We present implementations as a degenerate form of *fair discrete systems* (FDS) [KP00]. A fairness-free FDS $\mathcal{D} : \langle V, \Theta, \rho \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of Boolean variables. We define a *state s* to be an interpretation of V. Denote by Σ the set of all states. Assertions over V and satisfaction of assertions are defined like in games.
- Θ : The *initial condition*. This is an assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$.

We define a *run* of the FDS $\mathcal{D}$ to be a maximal sequence of states $\sigma : s_0, s_1, ...$, satisfying the requirements of

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For every $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

The sequence σ being maximal means that either σ is infinite, or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor.

We say that an FDS $\mathcal{D}$ *implements* specification φ if every run of $\mathcal{D}$ is infinite, and every run of $\mathcal{D}$ satisfies φ.

3.2 μ-calculus and Games

In [KPP05], we consider the case of GR[1] games (called there *generalized Streett[1]* games). In these games the winning condition is an implication between conjunctions of recurrence formulas (`always eventually!` φ where φ is a Boolean formula). These are exactly the types of goals in the games we defined in Section 3.1. We show how to solve such games in cubic time [KPP05]. We re-explain here how to compute the winning regions of each of the players and explain how to use the algorithm to extract a winning strategy. We start with a definition of μ-calculus over game structures. We give the μ-calculus formula that characterizes the set of winning states of the system. We explain how we construct from this μ-calculus formula an algorithm to compute the set of winning states. Finally, by saving intermediate values in the computation, we can construct a winning strategy and synthesize an FDS that implements the goal.

μ-calculus over Games Structures

We define μ-calculus [Koz83] over game structures. Let G: $\langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi\rangle$ be a GS. For every variable $v \in V$ the formulas v and $\neg v$ are *atomic formulas*. Let $Var = \{X, Y, \ldots\}$ be a set of *relational variables*. The μ-calculus formulas are constructed as follows.

$$\varphi ::= v \mid \neg v \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc\!\!\!\!\!\!\Diamond\, \varphi \mid \bigcirc\!\!\!\!\!\!\Box\, \varphi \mid \mu X \varphi \mid \nu X \varphi$$

A formula ψ is interpreted as the set of G-states in Σ in which ψ is true. We write such set of states as $[[\psi]]^e_G$ where G is the GS and $e : Var \rightarrow 2^\Sigma$ is an *environment*. The environment assigns to each relational variable a subset of Σ. We denote by $e[X \leftarrow S]$ the environment such that $e[X \leftarrow S](X) = S$ and $e[X \leftarrow S](Y) = e(Y)$ for $Y \neq X$. The set $[[\psi]]^e_G$ is defined inductively as follows[2].

- $[[v]]^e_G = \{s \in \Sigma \mid s[v] = 1\}$
- $[[\neg v]]^e_G = \{s \in \Sigma \mid s[v] = 0\}$
- $[[X]]^e_G = e(X)$
- $[[\varphi \vee \psi]]^e_G = [[\varphi]]^e_G \cup [[\psi]]^e_G$
- $[[\varphi \wedge \psi]]^e_G = [[\varphi]]^e_G \cap [[\psi]]^e_G$
- $[[\bigcirc\!\!\!\!\!\!\Diamond\, \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall \vec{x}',\ (s, \vec{x}') \models \rho_e \rightarrow \exists \vec{y}' \text{ such that } (s, \vec{x}', \vec{y}') \models \rho_s \\ \text{and } (\vec{x}', \vec{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\bigcirc\!\!\!\!\!\!\Diamond\, \varphi]]^e_G$ if the system can force the play to reach a state in $[[\varphi]]^e_G$. That is, regardless of how the environment moves from s, the system can choose an appropriate move into $[[\varphi]]^e_G$.
- $[[\bigcirc\!\!\!\!\!\!\Box\, \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \exists \vec{x}' \text{ such that } (s, \vec{x}') \models \rho_e \text{ and} \\ \forall \vec{y}',\ (s, \vec{x}', \vec{y}') \models \rho_s \rightarrow (\vec{x}', \vec{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\bigcirc\!\!\!\!\!\!\Box\, \varphi]]^e_G$ if the environment can force the play to reach a state in $[[\varphi]]^e_G$. As the environment moves first, it chooses an input $\vec{x}' \in X$ such that for all choices of the system the successor s is in $[[\varphi]]^e_G$.
- $[[\mu X \varphi]]^e_G = \cup_i S_i$ where $S_0 = \emptyset$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$
- $[[\nu X \varphi]]^e_G = \cap_i S_i$ where $S_0 = \Sigma$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$

When all the variables in φ are bound by either μ or ν the initial environment is not important and we simply write $[[\varphi]]_G$. In case that G is clear from the context we simply write $[[\varphi]]$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. A μ-calculus formula defines a symbolic algorithm for computing $[[\varphi]]$ [EL86]. For a μ-calculus formula of alternation depth k, the run time of this algorithm is $O(|\Sigma|^k)$. For a full exposition of μ-calculus we refer the reader to [Eme97]. We often abuse notations and write a μ-calculus formula φ instead of the set $[[\varphi]]$.

[2] Only for finite game structures.

In some cases, instead of using a very complex formula, it may be more readable to use *vector notation* as in Equation (2) below.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \begin{bmatrix} \mu Y(\Diamond Y \ \vee \ p \wedge \Diamond Z_2) \\ \mu Y(\Diamond Y \ \vee \ q \wedge \Diamond Z_1) \end{bmatrix} \tag{2}$$

Such a formula, may be viewed as the mutual fixpoint of the variables Z_1 and Z_2 or equivalently as an equal formula where a single variable Z replaces both Z_1 and Z_2 and ranges over pairs of states [Lic91]. The formula above characterizes the set of states from which system can force the game to visit p-states infinitely often and q-states infinitely often. We can characterize the same set of states by the following 'normal' formula[3].

$$\varphi = \nu Z\left([\mu Y(\Diamond Y \ \vee \ p \wedge \Diamond Z)] \ \wedge \ [\mu Y(\Diamond Y \ \vee \ q \wedge \Diamond Z)]\right)$$

Solving GR[1] Games

Let G be a game where the winning condition is of the following form.

$$\varphi : \quad \bigwedge_{i=1}^{m} \texttt{always eventually!}\, J_i^1 \implies \bigwedge_{j=1}^{n} \texttt{always eventually!}\, J_j^2$$

Here $\{J_i^1\}$ and $\{J_j^2\}$ are Boolean formulas. In [KPP05] we refer to these games as generalized Streett[1] games and provide the following μ-calculus formula to solve them. Let $j \oplus 1 = (j \ mod \ n) + 1$.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{i=1}^{m} \nu X(J_1^2 \wedge \Diamond Z_2 \ \vee \ \Diamond Y \ \vee \ \neg J_i^1 \wedge \Diamond X) \right) \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X(J_2^2 \wedge \Diamond Z_3 \ \vee \ \Diamond Y \ \vee \ \neg J_i^1 \wedge \Diamond X) \right) \\ \vdots \\ \vdots \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X(J_n^2 \wedge \Diamond Z_1 \ \vee \ \Diamond Y \ \vee \ \neg J_i^1 \wedge \Diamond X) \right) \end{bmatrix} \tag{3}$$

Intuitively, for $j \in [1..n]$ and $i \in [1..m]$ the greatest fixpoint $\nu X(J_j^2 \wedge \Diamond Z_{j\oplus 1} \ \vee \ \Diamond Y \ \vee \ \neg J_i^1 \wedge \Diamond X)$ characterizes the set of states from which the system can force the play either to stay indefinitely in $\neg J_i^1$ states (thus violating the left hand side of the implication) or in a finite number of steps reach a state in the set $J_j^2 \wedge \Diamond Z_{j\oplus 1} \ \vee \ \Diamond Y$. The two outer fixpoints make sure that the system wins from the set $J_j^2 \wedge \Diamond Z_{j\oplus 1} \ \vee \ \Diamond Y$. The least fixpoint μY makes sure that the unconstrained phase of

[3] This does not suggest a canonical translation from vector formulas to plain formulas. The same translation works for the formula in Equation (3) below. Note that the formula in Equation (2) and the formula in Equation (3) have a very similar structure.

a play represented by the disjunct $\diamondsuit Y$ is finite and ends in a $J_j^2 \wedge \diamondsuit Z_{j\oplus 1}$ state. Finally, the greatest fixpoint νZ_j is responsible for ensuring that, after visiting J_j^2, we can loop and visit $J_{j\oplus 1}^2$ and so on. By the cyclic dependence of the outermost greatest fixpoint, either all the sets in J_j^2 are visited or getting stuck in some inner greatest fixpoint, where some J_i^1 is visited only finitely many times.

We include below a (slightly simplified) code of the implementation of this μ-calculus formula in TLV. We denote J_i^α for $\alpha \in \{1,2\}$ by $Ji(i,\alpha)$ and $\diamondsuit$ by *cpred* (standing for *controlled predecessor*). The TLV statement `Fix(exp) S End` iterates statement S until the values of the expression `exp` at the end of two successive iterations are identical. Statement `S` is iterated at least twice. The `Fix` statement can be used to compute both a least- and a greatest-fixpoint, depending on the initial value of `exp`.

```
Func winm(m, n);
 Let z := 1;
 Fix(z)  -- Maximal fixpoint on z
   For (j in 1...n)
     Let r := 1;
     Let y := 0;
     Fix (y) -- Minimal fixpoint on y
       Let start := Ji(j,2) & cpred(z) | cpred(y);
       Let y := 0;
       For (i in 1...m)
         Let x := z;
         Fix (x) -- Maximal fixpoint within z on x
           Let x := start | !Ji(i,1) & cpred(x);
         End -- Fix (x)
         Let x[j][r][i] := x; // store values of x
         Let y := y | x;
       End -- For (i in 1...m)
       Let y[j][r] := y; // store values of y
       Let r := r + 1;
     End -- Fix (y)
     Let z := y;
     Let maxr[j] := r - 1;
   End -- For (j in 1...m)
 End -- Fix(z)
 Return z;
End -- Func winm(m, n);
```

We use the sets $y[j][r]$ and their subsets $x[j][r][i]$ to define N strategies for the system. The strategy f_j is defined on the states in Z_j. We show that the strategy f_j either forces the play to visit J_j^2 and then proceed to $Z_{j\oplus 1}$, or eventually avoid some J_i^1. We show that by combining these strategies, either the system switches strategies infinitely many times and ensures that the play be winning according to right hand side of the implication or eventually uses a fixed strategy ensuring that the play does not satisfy the left hand side of the implication. Essentially, the strategies are "go to $y[j][r]$ for minimal r" until getting to a J_j^2 state and then switch to strategy $j \oplus 1$ or "stay in $x[j][r][i]$".

It follows that we can solve realizability of LTL formulas in the form that interests us in polynomial (cubic) time.

Theorem 4. [KPP05] *Given sets of variables $\mathcal{X}$, $\mathcal{Y}$ whose set of possible valuations is Σ and an LTL formula φ with m and n conjuncts, we can determine using a symbolic algorithm whether φ is realizable in time proportional to* $(nm|\Sigma|)^3$.

3.3 Synthesis

We show how to use the intermediate values in the computation of the fixpoint to produce an FDS that implements φ. The FDS basically follows the strategies explained above.

Let $\mathcal{X}$, $\mathcal{Y}$, and φ be as above. Let $G\colon \langle \mathcal{X} \cup \mathcal{Y}, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \Theta, \varphi_g \rangle$ be the GS defined by $\mathcal{X}$, $\mathcal{Y}$, and φ. We construct the following FDS. Let $\mathcal{D} : \langle V_{\mathcal{D}}, \mathcal{X}, \mathcal{Y}_{\mathcal{D}}, \Theta_{\mathcal{D}}, \rho \rangle$ where $V_{\mathcal{D}} = V \cup \{jx\}$ and jx ranges over $[1..n]$, $\mathcal{Y}_{\mathcal{D}} = \mathcal{Y} \cup \{jx\}$, $\Theta_{\mathcal{D}} = \Theta \wedge (jx = 1)$. The variable jx is used to store internally which strategy should be applied. The transition ρ is the disjunction of the following three transitions:

Transition ρ_1 is the transition taken when a J^2_j state is reached and we change strategy from f_j to $f_{j\oplus 1}$. Accordingly, all the disjuncts in ρ_1 change jx. Transition ρ_2 is the transition taken in the case that we can get closer to a J^2_j state. These transitions go from states in some set $y[j][r]$ to states in the set $y[j][r']$ where $r' < r$. We take care to apply this transition only to states s for which $r > 1$ is the minimal index such that $s \in y[j][r]$. Transition ρ_3 is the transition taken from states $s \in x[j][r][i]$ such that $s \models \neg J^1_i$ and the transition takes us back to states in $x[j][r][i]$. Repeating such a transition forever will also lead to a legitimate computation because it violates the environment requirement of infinitely many visits to J^1_i-states. Again, we take care to apply this transition only to states for which (r,i) are the (lexicographically) minimal indices such that $s \in x[j][r][i]$.

Let $y[j][< r]$ denote the set $\bigcup_{l\in[1..r-1]} y[j][l]$. We write $(r',i') \prec (r,i)$ to denote that the pair (r',i') is lexicographically smaller than the pair (r,i). That is, either $r' < r$ or $r' = r$ and $i' = i$. Let $x[j][\prec(r,i)]$ denote the set $\bigcup_{(r',i')\prec(r,i)} x[j][r'][i']$. The transitions are defined as follows.

$$
\begin{aligned}
\rho_1 &= \bigvee_{j\in[1..n]} (jx{=}j) \wedge z \wedge J^2_j \wedge \rho_e \wedge \rho_s \wedge z' \wedge (jx'{=}j{\oplus}1)\\
\rho_2(j) &= \bigvee_{r>1} y[j][r] \wedge \neg y[j][< r] \wedge \rho_e \wedge \rho_s \wedge y'[j][< r]\\
\rho_2 &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_2(j)\\
\rho_3(j) &= \bigvee_{r} \bigvee_{i\in[1..m]} x[j][r][i] \wedge \neg x[j][\prec(r,i)] \wedge \neg J^1_i \wedge \rho_e \wedge \rho_s \wedge x'[j][r][i]\\
\rho_3 &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_3(j)
\end{aligned}
$$

The conjuncts $\neg y[j][< r]$ and $\neg x[j][\prec(r,i)]$ appearing in transitions $\rho_2(j)$ and $\rho_3(j)$ ensure the minimality of the indices to which these transitions are respectively applied.

Notice that the above transitions can be computed symbolically. We include below the TLV code that symbolically constructs the transition relation of the synthesized FDS and places it in `trans`. We denote the conjunction of ρ_e and ρ_s by `trans12`.

```
To symb_strategy;
  Let trans := 0;
  For (j in 1...n)
    Let jp1 := (j mod n) + 1;
    Let trans := trans | (jx=j) & z & Ji(j,2) & trans12 &
                          next(z) & (next(jx)=jp1);
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := y[j][1];
    For (r in 2...maxr[j])
      Let trans := trans | (jx=j) & y[j][r] & !low &
                      trans12 & next(low) & (next(jx)=j);
      Let low := low | y[j][r];
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := 0;
    For (r in 2...maxr[j])
      For (i in 1...m)
        Let trans := trans | (jx=j) & x[j][r][i] & !low
                                   & !ji(i,1) & trans12 &
                          next(x[j][r][i]) & (next(jx)=j);
        Let low := low | x[j][r][i];
      End -- For (i in 1...m)
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
End -- To symb_strategy;
```

Minimizing the Strategy

We have created an FDS that implements an LTL goal φ. The set of variables of this FDS includes the given set of input and output variables as well as a 'memory' variable jx. We have quite a liberal policy of choosing the next successor in the case of a visit to J^2_j. We simply choose some successor in the winning set. Here we minimize (symbolically) the resulting FDS. A necessary condition for the soundness of this minimization is that the specification be insensitive to stuttering.

Notice, that our FDS is deterministic. For every state and every possible assignment to the variables in $\mathcal{X} \cup \mathcal{Y}$ there exists at most one successor state with this assignment. Thus, removing transitions seems to be of lesser importance. We concentrate on removing redundant states.

As we are using the given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ the only possible candidate states for merging are states that agree on the values of variables in $\mathcal{X} \cup \mathcal{Y}$ and disagree on the value of jx. If we find two states s and s' such that $\rho(s,s')$, $s[\mathcal{X} \cup \mathcal{Y}] = s'[\mathcal{X} \cup \mathcal{Y}]$, and $s'[jx] = s[jx] \oplus 1$, we remove state s. We direct all its incoming arrows to s' and remove its outgoing arrows. Intuitively, we can do that because for every computation that passes through s there exists a computation that stutters once in s (due to the assumption of stuttering insensitivity). This modified computation passes from s to s' and still satisfies all the requirements (we know that stuttering in s is allowed because there exists a transition to s' which agrees with s on all variables).

As mentioned this minimization is performed symbolically. As we discuss in Section 3.4, it turns out that the minimization actually increases the size of the resulting BDDs. It seems to us that for practical reasons we may want to keep the size of BDDs minimal rather than minimize the automaton. The symbolic implementation of the minimization is given below. The transition *obseq* includes all possible assignments to V and V' such that all variables except jx maintain their values. It is enough to consider the transitions from j to $j \oplus 1$ for all j and then from n to j for all j to remove all redundant states. This is because the original transition just allows to increase jx by one.

```
For (j in 1..n)
  Let nextj := (j mod n)+1;
  reduce(j,nextj);
End -- For (j in 1..n)

For (j in 1..n-1)
  reduce(n,j)
End -- For (j in 1..n-1)

Func reduce(j,k)
  Let idle := trans & obseq & jx=j & next(jx)=k;
  Let states := idle forsome next(V);
  Let add_trans :=
      ((trans & next(states) & next(jx)=j) forsome jx) &
      next(jx)=k;
  Let rem_trans := next(states) & next(jx)=j1 |
                   states & jx=j1;
  Let add_init := ((init & states & jx=j1) forsome jx) &
                  jx=k;
  Let rem_init := states & jx=j;
  Let trans := (trans & !rem_trans) | add_trans;
  Let init := (init & !rem_init) | add_init;
  Return;
End -- Func reduce(j,k)
```

3.4 Example

As an example we consider the case of an arbiter. Our arbiter has n input lines in which clients request permissions and n output lines in which the clients are granted permission. We assume that initially the requests are set to zero, once a request has been made it cannot be withdrawn, and that the clients are fair, that is once a grant to a certain client has been given it eventually releases the resource by lowering its request line. Formally, the assumption on the environment in LTL format is below.

$$\bigwedge_i (\overline{r_i} \wedge \texttt{always}\ ((r_i{\neq}g_i) \rightarrow (r_i{=}\texttt{next!}\ r_i)) \wedge \texttt{always}\ ((r_i \wedge g_i) \rightarrow \texttt{eventually!}\ \overline{r_i}))$$

We expect the arbiter to initially give no grants, give at most one grant at a time (mutual exclusion), give only requested grants, maintain a grant as long as it is requested, to supply (eventually) every request, and to take grants that are no longer needed. Formally, the requirement from the system in LTL format is below.

$$\bigwedge_{i \neq j} \texttt{always}\ \neg(g_i \wedge g_j) \wedge \bigwedge_i \left(\overline{g_i} \wedge \left\{ \begin{array}{lcll} \texttt{always}\ ((r_i{=}g_i) & \rightarrow & (g_i{=}\texttt{next!}\ g_i)) & \wedge \\ \texttt{always}\ ((r_i \wedge \overline{g_i}) & \rightarrow & \texttt{eventually!}\ g_i) & \wedge \\ \texttt{always}\ ((\overline{r_i} \wedge g_i) & \rightarrow & \texttt{eventually!}\ \overline{g_i}) & \end{array} \right] \right)$$

The resulting game is $G\colon \langle V, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \varphi \rangle$ where

- $\mathcal{X} : \{r_i \mid i = 1, \ldots, n\}$
- $\mathcal{Y} : \{g_i \mid i = 1, \ldots, n\}$
- $\Theta : \bigwedge_i (\overline{r_i} \wedge \overline{g_i})$
- $\rho_e : \bigwedge_i ((r_i{\neq}g_i) \rightarrow (r_i'{=}r_i))$
- $\rho_s : \bigwedge_{i \neq j} \neg(g_i' \wedge g_j') \wedge \bigwedge_i ((r_i{=}g_i) \rightarrow (g_i' = g_i))$
- $\varphi :$

$$\begin{array}{l} \bigwedge_i \texttt{always}\ ((r_i \wedge g_i) \rightarrow \texttt{eventually!}\ r_i) \quad \rightarrow \\ \qquad\qquad \bigwedge_i \left[\begin{array}{ll} \texttt{always}\ ((r_i \wedge g_i) \rightarrow \texttt{eventually!}\ g_i) & \wedge \\ \texttt{always}\ ((\overline{r_i} \wedge g_i) \rightarrow \texttt{eventually!}\ \overline{g_i}) & \end{array} \right] \end{array}$$

We simplify φ by replacing “$\texttt{always}\ ((r_i \wedge g_i) \rightarrow \texttt{eventually!}\ \overline{r_i})$” by the formula “$\texttt{always eventually!}\ \neg(r_i \wedge g_i)$” and replacing the two formulas “$\texttt{always}\ ((r_i \wedge \overline{g_i}) \rightarrow \texttt{eventually!}\ g_i)$” and “$\texttt{always}\ ((\overline{r_i} \wedge g_i) \rightarrow \texttt{eventually!}\ \overline{g_i})$” by the single formula “$\texttt{always eventually!}\ (r_i{=}g_i)$”. This results in the simpler goal:

$$\varphi : \bigwedge_i \texttt{always eventually!}\ \neg(r_i \wedge g_i) \rightarrow \bigwedge_i \texttt{always eventually!}\ (r_i{=}g_i)$$

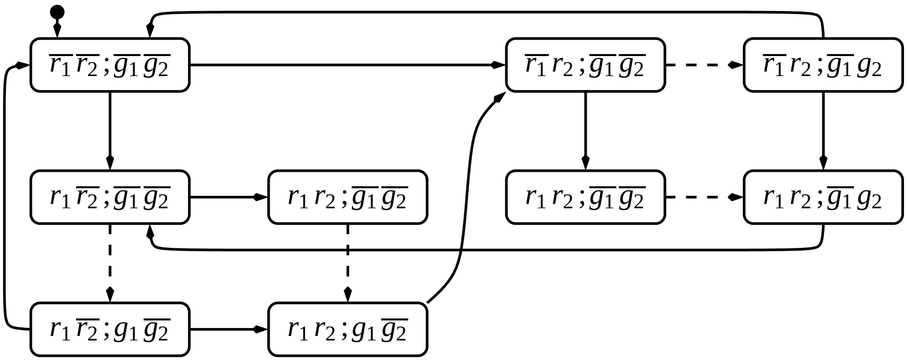
$\overline{r_1}\,\overline{r_2};\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,r_2;\overline{g_1}\,\overline{g_2}$
$\overline{r_1}\,r_2;\overline{g_1}\,g_2$
$r_1\,\overline{r_2};\overline{g_1}\,\overline{g_2}$
$r_1\,r_2;\overline{g_1}\,\overline{g_2}$
$r_1\,r_2;\overline{g_1}\,\overline{g_2}$
$r_1\,r_2;\overline{g_1}\,g_2$
$r_1\,\overline{r_2};g_1\,\overline{g_2}$
$r_1\,r_2;g_1\,\overline{g_2}$

Figure 6: Arbiter for 2

3.5 Extensions

The class of specifications to which the N^3-synthesis algorithm is applicable is wider than the limited form presented in Equation (1). The algorithm can be applied to any specification of the form $(\bigwedge_{i=1}^{m} \varphi_i) \implies (\bigwedge_{j=1}^{n} \psi_j)$, where each φ_i and ψ_j can be specified by an LTL formula of the form `always` `eventually!` q for a past formula q. Equivalently, each φ_i and ψ_j should be specifiable by a deterministic Büchi automaton. This is, for example, the case of the original version of the Arbiter, where the liveness conjuncts were each a response formula of the form `always` $(p \implies$ `eventually!` $q)$.

The way we deal with such a formula is to add to the game additional variables and a transition relation which encodes the deterministic Büchi automaton. For example, to deal with a formula `always` $(p \implies$ `eventually!` $q)$, we add to the game variables a new Boolean variable x with initial condition $x = 1$, and add to the transition relation ρ_e the additional conjunct

$$x' = (q \vee x \wedge \neg p)$$

We then replace in the specification the formula "`always` $(p \implies$ `eventually!` $q)$" by the conjunct "`always` `eventually!` x". It is not difficult to see that this is a sound transformation. That is, the formula `always` $(p \implies$ `eventually!` $q)$ is satisfied by a sequence σ iff there exists an interpretation of the variable x which satisfies the added transition relation and also equals 1 infinitely many times.

4 Complete Approach

In this section we describe in detail a general approach for synthesizing specifications given as arbitrary formulas in the linear-time segment of PSL. It was recently proposed by Kupferman and Vardi in [KV05] for LTL. We summarize the approach by applying it to the linear-time segment of PSL. The approach avoids Safra's intricate construction and nondeterministic parity tree automata, which are both necessary for the standard automata-theoretic approach to synthesis. Instead it uses a sequence of relatively simple automata translations. The complexity bound is doubly exponential but each translation leaves space for optimization. The approach works as follows. First, a nondeterministic Büchi word automaton for the negation of the specification is constructed and translated into a universal co-Büchi tree automaton that recognizes all trees containing only paths that satisfy the specification. This translation demands the first exponent of the complexity bound. Then, the tree automaton is translated into an alternating weak tree automaton, from which a nondeterministic Büchi tree automaton is built. The latter translation causes the second exponential blow-up. Finally, language emptiness for this nondeterministic Büchi automaton is computed. If the language is not empty the computation provides a witness, which corresponds to a correct implementation of the given specification.

In Section 4.1 we start by giving some formal definitions. Section 4.2 describes the translation of a PSL formula to a universal co-Büchi tree automaton. In Section 4.3 and 4.4 we describe the necessary automata translations. For the proofs we refer to [KV05]. Finally, we describe how to compute language emptiness and obtain the corresponding implementation.

4.1 Preliminaries

For a set D of directions, a D-tree is defined as a set $T \subseteq D^*$ such that if $x \cdot c \in T$ where $x \in D^*$, then also $x \in T$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For every $x \in T$, the nodes $x \cdot c$, for $c \in D$, are the *successors* of x. If $T = D^*$, we say T is a full D-tree. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or it has a unique successor. Given an alphabet Σ, a *Σ-labeled D-tree* is a pair (T, τ) where T is a tree and $\tau : T \rightarrow \Sigma$ maps each node of T to a letter in Σ.

For a set X, let $B^+(X)$ be the set of Boolean formulas build from elements in X using $\wedge$ and $\vee$ including the formulas $\mathbf{true}$ and $\mathbf{false}$. Given a set $Y \subseteq X$ and a formula $\theta \in B^+(X)$, we say that Y *satisfies* θ iff assigning $\mathbf{true}$ to elements in Y and assigning $\mathbf{false}$ to elements in $X \setminus Y$ makes θ true.

An *alternating tree automaton* is a tuple $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, where Σ is the input alphabet, D is a set of directions, Q is a finite set of states, q_0 is the initial state, $\delta : Q \times \Sigma \rightarrow B^+(D \times Q)$ is the transition function, and α is the acceptance condition, which defines a subset of Q^ω.

The alternating tree automaton A runs on a Σ-labeled full D-tree (T, τ). The tree is accepted if there exists a run on A which is accepting. A run r on A is a $(T \times Q)$-labeled $\mathbb{N}$-tree, respecting the transition relation [MS87]. A run is accepting if all its infinite paths satisfy the acceptance condition. Let us recall the acceptance conditions from Section 2.1. The set of states that are visited infinitely often on a path π are called $inf(\pi)$. A path π satisfies a *Büchi acceptance condition* $\alpha \subseteq Q$ if and only if $inf(\pi) \cap \alpha \neq \emptyset$. A path π satisfies a *co-Büchi acceptance condition* $\alpha \subseteq Q$ if and only if $inf(\pi) \cap \alpha = \emptyset$. Intuitively, a tree automaton and an input tree can be seen as two-player game where in each step the first player picks a transition according to the node in the input tree and the second player choose the direction.

In the following we describe special types of alternating automaton. If all the formulas that appear in δ are disjunctions of states, the automaton is called *nondeterministic*. Dually, if all the formulas are conjunctions it is called *universal*. An automaton with only one direction ($|D| = 1$) is called *word* automaton. An alternating *weak* automaton is an alternating Büchi automaton with a special structure of its state space. The states of an alternating weak automaton are partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that in each transition, the automaton either stays at the same set or moves to a set smaller in the partial order. Thus, each run of an alternating weak automaton eventually gets trapped in some set in the partition.

We denote the different types of automata by an acronym of three letters, where the first letter describes the branching mode (nondeterministic N, universal U, alternating A), the second letter describes the acceptance condition (Büchi B, co-Büchi C, Weak W), and the third letter describes the object on which the automaton runs (words W, trees T).

4.2 PSL to UCT

Given a set of properties written as a (linear-time) PSL formula ψ and a partitioning of the atomic proposition into input I and output signals O, we aim to find a strategy $f : (2^I)^* \rightarrow 2^O$ such that all the computations of the system generated by f satisfy ψ. Formally, a computation $\rho \in (2^{I \cup O})^\omega$ is generated by f if $\rho = (i_0 \cup o_0), (i_1 \cup o_1), (i_2 \cup o_2), \ldots$ and for all $j \geq 1 : o_j = f(i_0 \cdot i_1 \cdots i_{j-1})$. A strategy can be seen as a 2^O-labeled 2^I-tree. We will define a UCT S_ψ such that S_ψ accepts a 2^O-labeled 2^I-tree (T, τ) iff it is a good strategy for ψ.

We construct a NBW $A_{\neg\psi} = (2^{I \cup O}, Q, q_0, \delta, \alpha)$ for $\neg\psi$ using the automata construction of [BDBF$^+$05]. $A_{\neg\psi}$ accepts exactly all the words in $(2^{I \cup O})^\omega$ that do not satisfy ψ. Using $A_{\neg\psi}$ we construct $S_\psi = (2^O, 2^I, Q, q_0, \delta', \alpha)$, where for every $q \in Q$ and $o \in 2^O$, we have

$$\delta'(q,o) = \bigwedge_{i \in 2^I} \bigwedge_{s \in \delta(q, i \cup o)} (i,s).$$

S_ψ accepts a 2^O-labeled 2^I-tree (T,τ) iff for all path $\varepsilon, i_0, i_0 \cdot i_1, i_0 \cdot i_1 \cdot i_2, \ldots$ of T, the infinite word $(i_0 \cup \tau(\varepsilon)), (i_1 \cup \tau(i_0)), (i_2 \cup \tau(i_0 \cdot i_1)), \ldots$ is not accepted by $A_{\neg\psi}$. So all the computations generated by τ satisfy ψ. If the language accepted by S_ψ is not empty, we can obtain a witness, which is precisely the 2^O-labeled 2^I-tree we are aiming for. In the following sections, we translate the UCT to an equivalent NBT on which we check for language non-emptiness.

4.3 UCT to AWT

Given a UCT $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ which accepts some tree, we construct an AWT A' over the same alphabet with $O(|Q|^2 \cdot 2^{|Q|})$ states, such that $L(A') \neq \emptyset$ iff $L(A) \neq \emptyset$.

Let $n = |Q|$ and $k = n \cdot 2^n$. We construct $A' = (\Sigma, D, Q', q_0', \delta', \alpha')$ with

$$\begin{array}{rcl}
Q' & = & Q \times \{0, 1, \ldots, 2k\}, \\
q_0' & = & (q_0, 2k), \\
\delta'((q,i),\sigma) & = & \begin{cases} \mathit{release}(\delta(q,\sigma), i) & \text{If } q \notin \alpha \text{ or } i \text{ is even,} \\ \mathbf{false} & \text{If } q \in \alpha \text{ and } i \text{ is odd,} \end{cases} \\
\alpha' & = & Q \times \{1, 3, \ldots, 2k-1\},
\end{array}$$

where $\mathit{release}(\theta, i)$ is obtained from θ by replacing an atom (c,q) by the disjunction $\bigvee_{i' \le i} (c, (q, i'))$.

4.4 AWT to NBT

The translation of AWT to NBW is a variant of the translation of ABW to NBW proposed by Miyano and Hayashi [MH84].

Given an AWT $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ with n states. There is an NBT A' with $2^{O(n)}$

states such that $L(A') = L(A)$. Let

$$\begin{array}{rcl} S'_c & = & \{s : (c,s) \in S'\}, \\ O'_c & = & \{o : (c,o) \in O'\}, \\ sat(S,\sigma) & \subseteq & 2^{D \times Q} \text{ includes all sets of } D \times Q \text{ that satisfy } \bigwedge\limits_{q \in S} \delta(q,\sigma), \text{ and} \\ (S',O') & \in & pairset(S,O,\sigma) \iff \left\{ \begin{array}{l} S' \in sat(S,\sigma), \\ O' \in sat(O,\sigma), \\ O' \subseteq S'. \end{array} \right. \end{array}$$

We define $A' = (\Sigma, D, Q', q'_0, \delta', \alpha')$ such that

$$\begin{array}{rcl} Q' & = & 2^Q \times 2^Q, \\ q'_0 & = & (\{q_0\}, \emptyset), \\ \delta'((S,O),\sigma) & = & \left\{ \begin{array}{lll} \bigvee_{(S',O') \in pairset(S,O,\sigma)} \bigwedge_{c \in D} & (c,(S'_c, O'_c \setminus \alpha)) & \text{If } O \neq \emptyset, \\ \bigvee_{S' \in sat(S,\sigma)} \bigwedge_{c \in D} & (c,(S'_c,\ S'_c \setminus \alpha)) & \text{If } O = \emptyset, \end{array} \right. \\ \alpha' & = & 2^Q \times \{\emptyset\}. \end{array}$$

Essentially, A' guesses a subset construction applied to a run of A. At a given node x of a run, it keeps in its memory the set of states in which the various copies of A visit nodes x in the guessed run. In order to make sure that every infinite path visits states in α infinitely often, A' keeps track of states that "owe" a visit to α.

4.5 Language Emptiness

Language emptiness checking of an NBT $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ is the same as solving an infinite two-player game G with Büchi acceptance condition. In each step Player 0 picks a transition which includes the choice of an input letter and Player 1 chooses a direction. The game proceeds as follows. Player 0 starts a game by picking an initial transition from δ and a corresponding input letter from Σ. Player 1 determines the successor to proceed with. His opponent reacts by again selecting a transition from δ, where the current state matches the chosen successor state of the previous transition. The sequence of actions represents a play of the game and induces an infinite sequence of states visited along the path across the tree. Player 0 wins the play if this infinite state sequence satisfies the acceptance condition of A, otherwise Player 1 wins. Player 1 tries to prevent Player 0 from being the winner, his goal is to verify the existence of a path such that the corresponding state sequence violates the acceptance condition of A. In the Büchi game corresponding to A, Player 0 has a winning strategy if and only if the language accepted by A is not empty. So the winning strategy represents a witness for the language accepted by A. In Büchi games the winning player has a memoryless winning strategy, which means that the decisions a player has to make to win a play depends only on the current state (no “memory of the past” is needed). We can easily compute this memoryless winning strategy [Tho95]. When we restrict the game to the computed winning strategy, we obtain a system, which answers correctly to all inputs sequences and we are done.

5 Conclusions

We have presented three approaches that tackle the problem of realizability and synthesis for the linear-time segment of PSL. The problem is known to be highly complex and the standard approach to synthesize LTL, which is similar to linear-time PSL, demands the intricate determinization construction of Safra. We have shown three different ways to control the complexity and avoid Safra's construction. All presented approaches have their own strengths and weaknesses and so they complement each other.

The nondeterministic synthesis approach is based on using nondeterministic Büchi automata to deal with this complex problem. The approach is very efficient but handles only a modest subset of PSL. We have applied the approach to completion of partially implemented systems, and in particular to program repair. The algorithm finds readable repairs in acceptable time, though improvements in the implementation are still possible.

Reactivity[1] synthesis solves realizability and synthesis for a substantial subset of LTL by avoiding the construction of an automaton. We also presented an algorithm which reduces the number of states in the synthesized module for the case that the specification is stuttering insensitive. We have shown that this approach can be applied to wide class of formulas, which covers the full set of generalized reactivity[1] properties, and illustrated that the language is sufficiently expressive to specify most specifications that arise in hardware designs.

The approach we presented last is a complete approach to synthesis of PSL and is based on an approach by [KV05]. Since the approach is complete, its complexity remains doubly exponential for LTL and correspondingly more complex for the more succinct Property Specification Language. However, it avoids Safra's determinization construction using several relatively simple automata translations.

These three approaches together form a palette from which the user can select the appropriate synthesis algorithm, depending on the type of properties that need to be synthesized. We have thus circumvented the need for expensive and complicated algorithms in most cases, while retaining the possibility of synthesizing more difficult PSL properties.

6 References

[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.

[AT04] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[BDBF^{+}05] Shoham Ben-David, Roderick Bloem, Dana Fisman, Andreas Griesmayer, Ingo Pill, and Sitvanit Ruah. Automata construction algorithms optimized for PSL, 2005. Prosyd D 3.2/4.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[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.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, Upsala, 1963.

[CKW05] R. Chen, D. Köb, and F. Wotawa. A comparison of fault explanation and localization. unpublished, 2005.

[EL86] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional modal μ-calculus. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 267–278, 1986.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. American Mathematical Society, 1997.

[FHW80] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:111–121, 1980.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[GV03] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Model Checking of Software: 10th International SPIN Workshop*, pages 121–135. Springer-Verlag, May 2003. LNCS 2648.

[Har05] A. Harding. *Symbolic Strategy Synthesis For Games With LTL Winning Conditions*. PhD thesis, University of Birmingham, 2005.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th symposium on Principles of programming languages (POPL'02)*, pages 58–70. ACM Press, 2002.

[HS96] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, MA, 1996.

[KMTV00] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th Int. Conf. on Concurrency Theory*, volume 1877 of *LNCS*, pages 92–107. Springer-Verlag, 2000.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 163:203–243, 2000.

[KPP05] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Comp.*, 200(1):36–61, 2005.

[KV98] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[KV05] O. Kupferman and M. Vardi. Safraless decision procedures. To appear at 39th IEEE Symposium on Foundations of Computer Science, 2005.

[Lic91] O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, Weizmann Institute of Science, 1991.

[Mai00] M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Annual Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[MS87] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[MSW00] C. Mateis, M. Stumptner, and F. Wotawa. A value-based diagnosis model for Java programs. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis*, 2000.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[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.

[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.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Church's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.

[Saf88] S. Safra. On the complexity of ω-automata. In *Proc. 29th IEEE Symp. Found. of Comp. Sci.*, pages 319–327, 1988. An extended version to appear in *J. Comp. Sys. Sci.*

[SB05] S. Staber and R. Bloem. Property-based fault localization, 2005. Prosyd D 2.2/2.

[ST03] R. Sebastiani and S. Tonetta. “More deterministic” vs. “smaller” Büchi automata for efficient LTL model checking. In *Correct Hardware Design and Verification Methods (CHARME’03)*, pages 126–140, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13. Springer-Verlag, 1995. LNCS 900.