Synthesis of Reactive Systems

Jens Regenberg
<regenberg@react.cs.uni-sb.de>

Diploma Thesis

Prof. Bernd Finkbeiner
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 – Informatik
Universität des Saarlandes, Saarbrücken, 2005

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Saarbrücken, December 4, 2005.

Jens Regenberg

Acknowledgements

I am indebted to many people who have helped me during my studies and in preparing this thesis. I want to thank Bernd Finkbeiner who offered Tobias Maurer and me the topic of our theses. Sven Schewe was a great tutor and patiently answered all questions. Special thanks to Tobias Maurer with whom I developed ReaSyn and of course to my parents Uschi and Uwe Regenberg and my fiancée Nina, who supported me througout my whole studies. I love you!

Contents

Abstract

Software development generally implies three phases, design, implementation, and verification. As the separation of implementation and verification is time-consuming and effortful, both phases can be replaced by an automated synthesis development phase. The problem of distributed synthesis is to examine whether for a specification and an architecture there exists an implementation that meets the specification. ReaSyn is a tool, which examines for a given architecture and specification whether the underlying distributed system is realizable and creates an according program. Existing theoretical approaches were implemented: ReaSyn constructs and solves distributed games [MW03] in order to generate an optimized PROMELA program. Finally, ReaSyn is evaluated with respect to experimental results and possible extensions are discussed.

Chapter 1

Introduction

Software development is a time-consuming task. Usually it follows three steps. First, all components of the piece of software are designed and their behavior is specified. The design phase is followed by the implementation of the system. Only after the implementation is finished, it is possible to verify the system, i.e., to prove that the implementation satisfies the specified behavior.

An alternative to the effortful tasks of implementation and subsequent verification is to synthesize the implementation. Using a given specification and architecture, synthesis combines implementation and verification in one automated development phase. The resulting program does not require an extensive verification, because due to the synthesis process, the correctness of the program is inherently given. Thus, given an architecture and a specification, synthesis allows to directly create a program, which fulfills the specification. The resulting implementation is defined as a set of finite state programs satisfying the specification. If the given system is not limited to one, but entails several processes, the synthesis is referred to as *distributed synthesis*. Besides automation, synthesis has other advantages. Unrealizable specifications are identified before time is wasted in the implementation process. Additionally, the debugging cycles, which may not even terminate in cases of unrealizable specifications, become obsolete.

The goal of this project was to develop a tool, ReaSyn, which examines whether distributed systems are realizable. Additionally, if they are realizable, a small implementation of the system should be generated. ReaSyn was realized in cooperation with Maurer [Mau05].

So far, research has not provided an implementation of the theoretical aproaches to distributed synthesis. Distributed synthesis is a problem with non-elementary complexity. A second aim of this project was hence to evaluate whether distributed synthesis is possible at all. If so, the goal was to identify its boundary conditions.

1.1 History of System Synthesis

Generally, it has been demonstrated that synthesis of reactive systems is possible, but so far, no implementation has been presented. Pnuelli and Rosner [PR90] have shown that the Distributed Synthesis Problem in general is not decidable. This insight was influenced by the work of Peterson and Reif [PR79] who studied multi-player games with incomplete information. But there are classes of architectures, for which the Distributed Synthesis Problem is proven to be decidable. Rosner [Ros92] approached the synthesis of distributed asynchronous systems with LTL (linear temporal logic) specifications. The synthesis problem for a single process architecture was solved for the specification logic CTL* (computational tree logic) [KV97, KV99] as well as for the μ-calculus [KV00]. Two-way pipelines and one-way ring architectures with CTL* specifications are proven to be decidable in [KV01]. Another approach to distributed synthesis was made by Madhusudan and Thiagarajan [MT01, MT02a, **?**]. They solved the synthesis problem for LTL in an asynchronous setting. Furthermore, Finkbeiner and Schewe found a criterion [FS05] by which one can efficiently decide whether for a given architecture the distributed synthesis problem is decidable.

There are several approaches to solve the Distributed Synthesis Problem. [PR90] synthesize a program for a single process architecture and subsequently decompose this program into distinct programs for all processes of the architecture. The algorithms of [FS05, KV01] use an automata-based construction. It starts with an automaton equivalent to the specification. In the following, the architecture is incorporated into this automaton. There exists an implementation if and only if the language of the resulting automaton is not empty. A game-theoretic approach is proposed in [MW03]. Here, the specification and the architectures are first encoded into two-player games of player versus environment. In an iterative procedure, the specification game and an architecture game are combined into a new game. The result is a single, solvable two-player game [GTW02]. If and only if there exists a winning strategy for the player, there also exists an implementation that satisfies the specification.

1.2 Our Approach

Even though the literature offers the aforementioned theoretical solutions, none of them has been actually implemented. Therefore, ReaSyn was created as a tool to solve instances of the Distributed Synthesis Problem. The problem of distributed synthesis is to examine whether for a specification φ and an architecture $\mathcal{A}$ there exists an implementation for $\mathcal{A}$ that satisfies φ.

A Distributed Synthesis Problem is constituted of an architecture and a specification. Architectures model the structure and the internal communication

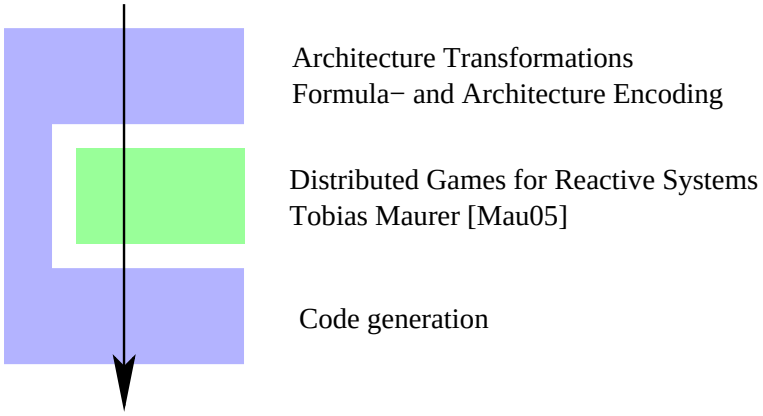


FIGURE 1.1 Interaction between the two theses

of the distributed system, while the specification defines its desired behavior. The efford needed to solve the Distributed Synthesis Problem depends on the sizes of the architecture and the specification. Finkbeiner and Schewe [FS05] introduced the "Information Fork" criterion, which identifies architectures for which the Distributed Synthesis Problem is decidable. For this decidable subclass, ReaSyn decides whether the distributed systems are realizable, given a specification in linear temporal logic (LTL).

In order to solve such instances of the Distributed Synthesis Problem ReaSyn takes a distributed games approach as proposed by Mohalik and Walukiewicz [MW03]. Due to the restrictions a distributed game imposes on the architectures, the architecture provided by the user is transformed into a pipeline architecture. Therefore, the transformations discussed in [FS05], which transform every architecture without an information fork into a strictly hierarchical acyclic architecture, are used. Finally, the architectures are converted into pipeline architectures. From the solution of the Distributed Synthesis Problem on pipeline architectures a solution for the originally provided architectures is derived.

Additionally, a transformation chain is introduced to convert the LTL specification provided by the user into a deterministic automaton. The transformation of the LTL specification uses the tool LTL2BA by Gastin and Oddoux [GO01] to convert the specification into a Büchi automaton. In order to reduce the size of the specification part in the distributed game, the Büchi automaton is optimized before further transformations. Finally, a two-player game simulating the deterministic automaton is used to construct the distributed game. Subsequently, the distributed game is step-wise reduced to a two-player game of Player versus Environment. If the Player wins the game, he has a winning strategy and the distributed system is realizable. The winning strategy is used to create a finite transition system for each process of the original architecture. It is represented as a PROMELA program.

ReaSyn was developed in cooperation with Maurer [Mau05]. He focused on the distributed games including the datastructures and algorithms to reduce and solve the distributed games. The present work covers architecture and specification transformations as well as the decomposition and optimization of strategies and the code generation. Additionally, it introduces an automata based transformation of winning conditions. Figure 1.1 gives an overview of ReaSyn's structure. The first part (Chapters 2 - 7) covers the underlying theory of the architecture and specification transformations. Next, ReaSyn and its implementation are introduced in detail (Chapters 8 - 10). Finally, the work is evaluated with respect to experimental results and possible extensions are discussed.

Part I

Theory

The following chapters formally define the Distributed Synthesis Problem and its components architecture and specification. The architectures describe the structure of the distributed system under consideration. They can be considered as a directed labled graph, where the processes are represented as nodes and the signals, by which the processes communicate, are the edges between the nodes. In every architecture there is a designated process, which simulates the environment in which the distributed system is executed. In fact, in a distributed system every process interacts with a different environment, but as a simplification all these different environment processes are combined. In context of ReaSyn the communication via the signals happens without delay. In a single atomic step of the system, all processes of the architecture read input from incoming signals and produce their output. Additionally, ReaSyn assumes that the processes of an architecture can be ordered according to their knowledge of the distributed system. Processes, which receive input directly from the environment process are the best informed processes. The less informed processes read their input from the processes with greater knowledge. The specification defines the desired behavior of the distributed system. It is given as an LTL formula. Generally, the behavior of a system is represented as a tree, where different branches denote different behaviors of the system. The different behaviors arouse due to varying input values on the signals under environment control. In order to check whether such a tree satisfies an LTL formula, the LTL formula is tested on every possible path of the tree representing the behavior of the distributed system.

In order to combine both the architecture and the specification into a distributed game [MW03] they have to be converted into two player parity games. If possible, the architecture is first transformed into a pipeline architecture. The resulting pipeline architecture is subsequently encoded into a parity game for each process of the pipeline. One player simulates the process' behavior; the other player chooses the input of the process. The transformation of the specification is more complex. It involves several automata and conversions between them. In the end, the specification is transformed into a deterministic parity automaton, which is encoded into a game simulating the automaton. One player chooses a letter from the automaton's alphabet, the other chooses the transition the automaton would take. In the distributed game, the architecture games are used to generate a behavior pattern of the distributed system and the specification game is used to check whether the generated pattern fulfills the specification.

Games are introduced in Chapter 3. The transformations used to convert an architecture into a set of two-player games are presented in Chapter 4, while the specification transformations are presented in Chapters 5 and 6. The following chapter covers the formal definition of the Distributed Synthesis Problem and its components.

Chapter 2

Preliminaries

2.1 Architectures

Architectures structure the way processes communicate with each other. The communication media are called signals. Signals can be written to and read by processes. Interpreting processes as nodes and signals as connecting edges allows to think of an architecture as a directed labled graph.

Definition 1: Architecture

> An Architecture is given as a tuple $\mathcal{A} = (P, S, E_S, p_{env}, I, O)$ where P is the set of processes and $p_{env} \in P$ is the designated environment process, S is the set of signals (variables). $I : P \to 2^S$ is a function that maps a process to the set of its input signals and analogously $O : P \to 2^S$ is a function that maps a process to its set of output signals. Furthermore, $S = \dot{\bigcup}_{p \in P} O(p)$; $E_S \subseteq P \times S \times P$ is the set of edges labeled with signals. Additionally, $(p, s, q) \in E_S \Leftrightarrow s \in O(p) \wedge s \in I(q)$.

In addition to the definition, the following notation is used: Architecture $\mathcal{A}$ is called acyclic if (P, E_S) is acyclic. A signal's domain $\mathrm{dom}(s), s \in S$ is the set of values s can hold. A process $p \in P$ is called idle if $O(p) = \emptyset$. $P^- = \{p \in P \setminus \{p_{env}\} \mid O(p) \neq \emptyset\}$ is the set of non idle processes. In order to avoid inconsistencies, we also assume that the sets $O(p)$ are pairwise disjoint ($\forall p_1, p_2 \in P. p_1 \neq p_2 \Rightarrow O(p_1) \cap O(p_2) = \emptyset$). This means that each signal can only be written to by a single process. However, there is no restriction on how many processes can read a signal. The set $O(p, q) = \{s \in S \mid (p, s, q) \in E_S\}$ is the set of signals, which label the edges between p and q. $READ : S \to 2^P$ returns the set of processes reading the given signal. The corresponding function $WRITE : S \to 2^P$ returns the set of processes, which are writing to the specified signal. Note that in our setup

$WRITE(s)$ is always a singleton set to avoid inconsistencies. Furthermore, the set $D(S') = \mathrm{dom}(s_1) \times \cdots \times \mathrm{dom}(s_n)$ is defined to be the cartesian product of signal domains, where $S' = \{s_1, \ldots, s_n\} \subseteq S$.

2.2 Syntax and Semantics of LTL

LTL formulas are used to describe the execution properties of a system. The set $atom$ of atomic propositions contains all properties that can be valid in a given state of the system. Using the boolean operators $(\vee, \wedge, \neg)$, one can express static properties. The temporal operators next (X), until (U), eventually ($\Diamond$) and henceforth ($\Box$) provide the possibility to state dynamic properties.

Definition 2: Syntax of LTL Formulas [GO01]

> The set of LTL formulas on the set $atom$ of atomic Propositions is defined by
>
> $$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\,\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$
>
> where $p \in atom$.

An LTL formula can be interpreted regarding different types of systems. Usually, it is interpreted with respect to a linear structure but it can be interpreted with regard to a tree-like structure as well. In a tree, every path resembles an execution behavior of the system. The LTL formula is interpreted over every possible path in the tree. A linear model can be seen as a unary tree. In case of a linear system one speaks of word semantics of LTL. The semantics, which interpret LTL formulas over arbitrary trees, are called tree semantics of LTL.

Definition 3: Σ-labeled Υ-tree

> A (full) tree is given as the set $\Upsilon^\star$ of all finite words over a set of directions Υ. Every non-empty node $x \cdot v, x \in \Upsilon^\star, v \in \Upsilon$, has the direction $\mathrm{dir}(x \cdot v) = v$ and the empty word ϵ has a designated root direction $\mathrm{dir}(\epsilon) = v_0 \in \Upsilon$. Given the finite sets Σ and Υ, the Σ-labeled Υ-tree is a pair $\langle \Upsilon^\star, l \rangle$, where $l : \Upsilon^\star \to \Sigma$ is a labeling function, which maps a node from $\Upsilon^\star$ to a letter of Σ.

Definition 4: Run of a System

> A run $\sigma = \sigma_1\sigma_2\cdots \in \Sigma^\omega$ of a system is a infinite sequence of system states. This sequence resembles a single execution of

the system. Each σ_i contains the valid atomic propositions of the system state.

The word semantics of LTL define whether or not a run σ of a given system satisfies a given formula φ. The semantics depend on the atomic propositions that are valid in the states of σ.

Definition 5: Word Semantics of LTL Formulas [GO01]

Let $\sigma = \sigma_0\sigma_1...$ be a word in Σ^ω with $\Sigma = 2^{atom}$ and φ an LTL formula. *atom* is the set of atomic propositions. The relation $\sigma \models \varphi$ (σ models φ) is defined as follows:

- $\sigma \models p$ iff $p \in \sigma_0$,
- $\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$,
- $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$,
- $\sigma \models \mathsf{X}\varphi$ iff $\sigma_1\sigma_2... \models \varphi$,
- $\sigma \models \varphi_1 \ \mathsf{U} \ \varphi_2$ iff $\exists k \geq 0. \sigma_k\sigma_{k+1}... \models \varphi_2$ and $\forall 0 \leq i < k. \sigma_i\sigma_{i+1}... \models \varphi_1$.

The additional operators are derived from the basic ones. They are defined by:

- $\texttt{true} \stackrel{def}{=} p \vee \neg p$
- $\texttt{false} \stackrel{def}{=} \neg\,\texttt{true}$
- $\varphi_1 \wedge \varphi_2 \stackrel{def}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\Diamond\varphi \stackrel{def}{=} \texttt{true U } \varphi$
- $\Box\varphi \stackrel{def}{=} \neg\Diamond(\neg\varphi)$

The size of an LTL formula φ is defined as the number of subformulas of φ.

Definition 6: Tree Semantics of LTL Formulas

Let $\pi = \upsilon_1\upsilon_2\cdots \in \Upsilon^\omega, \upsilon_i \in \Upsilon^\star$ be a path through the Σ-labeled Υ-tree $\langle \Upsilon^\star, l\rangle$. The relation $\models$ is defined by:

$$\langle \Upsilon^\star, l\rangle \models \varphi \stackrel{\text{def.}}{\Longleftrightarrow} \forall \upsilon_1\upsilon_2\cdots \in \Upsilon^\omega. l(\upsilon_1)l(\upsilon_2)\cdots \models \varphi.$$

2.3 Partition of the Distributed Synthesis Problem

Given the above mentioned definitions, one can formally define the Distributed Synthesis Problem:

Definition 7: Distributed Synthesis Problem

> For a given architecture $\mathcal{A}$ and an LTL formula φ, the Distributed Synthesis Problem $(\mathcal{A}, \varphi)$ is the problem to decide, whether there exists an implementation for $\mathcal{A}$, which satisfies φ. Additionally, if there exists such an implementation, the problem extends to finding one of them.

ReaSyn addresses the Distributed Synthesis Problem in three parts:

1. Part one, covered in this thesis, handles the transformations needed to convert the specification and the architecture into games. Furthermore, it provides functionality to convert the "all-path parity" condition (cf. Chapter 7) into a parity condition.
2. The second part, covered in [Mau05], is concerned with solving a distributed game [MW03].
3. Finally, the third part covers the optimization and generation of PROMELA code for the distributed system. It is also covered in this thesis.

A solution to an instance of the Distributed Synthesis Problem is a tree, which satisfies the specification and represents an implementation of the architecture processes. The external input values of the system span this tree; every path starting at the root constitutes a run of the system. Part two creates these trees by combining the architecture games and the specification game. These games are constructed by the first part. The specification game is used to check whether every path in the tree fulfills the specification, while the architecture games create the branches of the tree. Finally, the third part chooses a tree, which solves the given instance of the Distributed Synthesis Problem and generates a PROMELA program induced by the tree.

In order to formalize the interaction of these parts with each other and the user, there are two interfaces. The parts of ReaSyn covered by this thesis interact with two parties, namely, the user and the part described in [Mau05]. The interface controlling the interaction with the user is covered in Chapter 8, while Chapter 9 describes the interface between parts one and two and parts two and three.

Chapter 3

Games

This chapter gives a short introduction to infinite two-player games on directed graphs. One player is called Player, the other is called Environment.

3.1 Infinite Games

A game consists of an arena and a winning condition.

Definition 8: Arenas [GTW02]

> An arena is a triple $\mathfrak{A} = (V_{player}, V_{env}, E)$, where V_{player} is a finite set of nodes belonging to the Player. V_{env} is the finite set of Environment nodes. The set of all nodes is denoted by $V = V_{player} \dot{\cup} V_{env}$. $E \subseteq V \times V$ is the transition relation.

In addition, let $vE = \{v' \in V \mid (v, v') \in E\}$ be the set of successors of v. For a player $\sigma \in \{Player, Environment\}$, $\bar{\sigma}$ determines the remaining player. A bipartite game is a game where Player and Environment nodes alternate on every path through (V, E) ($\forall v \in V_\sigma . vE \subseteq V_{\bar{\sigma}}$).

One play of a game can be visualised as follows. A token is placed on an initial node v_0. If v_0 is a Player node, then the Player moves the token from v_0 to a successor node $v' \in v_0E$. Analogously, if v_0 is an Environment node, the environment moves the token to a successor. Formally, if $v \in V$ is a $\sigma \in \{Player, Environment\}$ node then σ chooses one successor $v' \in vE$. This procedure is carried out until the token reaches a dead end, i.e., a node v with $vE = \emptyset$.

Definition 9: Play [GTW02]

A play in the Arena $\mathfrak{A}$ is defined either as

- an infinite path $\pi = v_0 v_1 v_2 \cdots \in V^\omega$ (**infinite play**)
- or a finite path $\pi = v_0 v_1 v_2 \ldots v_n \in V^+$ (**finite play**)

with $\forall i \in \mathbb{N}.v_{i+1} \in v_i E$ in case of an infinite play or $v_n E = \emptyset \wedge \forall 0 \leq i < n.v_{i+1} \in v_i E$ in case of a finite play. For an infinite play π the infinity set $\mathrm{Inf}(\pi)$ is defined to be the set of nodes that occur infinitly often in π

Winning conditions for games are the same as the winning conditions for automata. Although there are many more, in the case of ReaSyn, Büchi-, Rabin-, parity, and "all-path" parity winning conditions, as defined below, are applicable. The other mentionend winning conditions appear in the context of LTL2BA [GO01] and will be discussed below. In analogy to the winning conditions for automata, parity winning conditions rely on a coloring function to categorize the existing nodes: let $\chi : V \to C$ be a coloring function that maps a node to a color. The coloring of a play $\chi(\pi)$ is defined as $\chi(v_0)\chi(v_1)\ldots$.

Definition 10: Winning Condition

Let ACC be an arbitrary winning condition. By $W(ACC)$ we denote the set of all infinite plays π, such that π is accepted by ACC. We define the following winning conditions [GTW02]:

- **Büchi condition** $ACC = F \subseteq V$: $\pi \in W(ACC)$ if and only if $\mathrm{Inf}(\pi) \cap F \neq \emptyset$
- **co-Büchi condition** $ACC \subseteq V$: $\pi \in W(ACC)$ if and only if $\mathrm{Inf}(\pi) \cap ACC = \emptyset$
- **Rabin condition** $ACC = \{(G_0, R_0), (G_1, R_1), \ldots (G_n, R_n)\}$: $\pi \in W(ACC)$ if and only if $\exists 0 \leq k \leq n$ such that $\mathrm{Inf}(\pi) \cap E_k = \emptyset \wedge \mathrm{Inf}(\pi) \cap F_k \neq \emptyset$.
- **minimal parity condition** ($\chi : V \to \mathbb{N}$): $\pi \in W(ACC) \Leftrightarrow \min(\mathrm{Inf}(\chi(\pi)))$ is even.
- **generalized Büchi** [GO01]: $ACC = T \subseteq V \times \Sigma \times V$. $\pi \in W(ACC) \Leftrightarrow \exists v^\sigma v' \in T$ that appears infinitly often in π. In other words, some transitions from T are taken infinitly often.

In anology to mininmal parity conditions there also is a maximal parity condition. There, the maximal parity occuring infinitly often has to be even. During the reduction of the distributed game another winning condition appears,

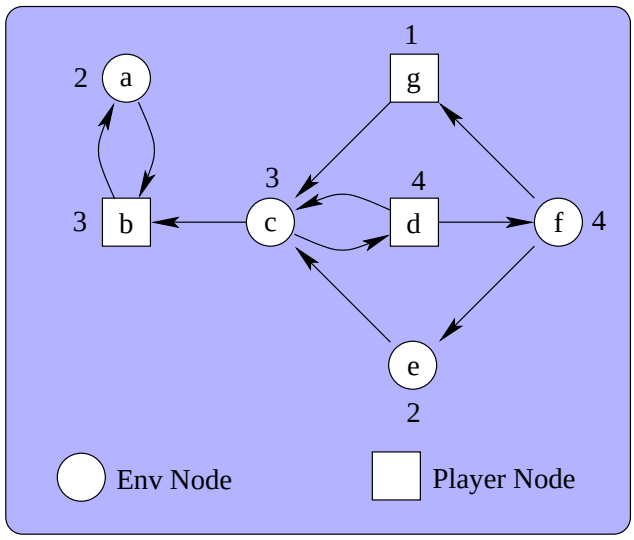


FIGURE 3.1 Colored Arena;
1,2,3,4 are colors

which is called an "all-path parity" condition. It is introduced in Chapter 7 as some additional definitions are needed.

Finally, a game can be defined:

Definition 11: Game [GTW02]

Let $\mathfrak{A}$ be an arena as above and $Win \subseteq V^\omega$. The pair

$$\mathcal{G} = (\mathfrak{A}, Win)$$

is called a game. A Tuple $(\mathcal{G}, v_0)$ is called an initialized game, if every play π begins with v_0. The Player is declared the winner of a play, if and only if

- $\pi = v_0 v_1 \dots v_n$ is a finite play and v_n is an Environment node with $v_n E = \emptyset$ or
- $\pi = v_0 v_1 \dots$ is an infinte play and $\pi \in Win$.

The Environment wins π, if the Player does not win π. If the Player wins an initialized game, he is called winner of the game.

3.2 Strategies and Trees

Having defined the possible winning conditions, one needs to examine, whether a player (Player or Environment) can win the game from a given node, regard-

less of how the other moves. This possibility is given if a player has a strategy leading him through the game in a way that he will win it. Consider the game in Figure 3.1 under the assumption of a min parity winning condition. From node b, Player can only move to node a. The Environment can only move the token back to b. In this cycle the Player wins with the strategy "*If I'm in b then I'll move the token to node a*". If the game starts in a different node, the Environment will win every play with the strategy: "*Whenever the token is in node c, I'll move it to node d and if it is in node f, I'll move it towards node g*". If the Environment moves the token from c to b instead of d, it will loose the play because then the Player "can" always stay in $\{a, b\}$. Furthermore, if the Environment moves the token from f to e instead of g, Player wins the min parity game. Formally, a strategy is:

Definition 12: Strategy [GTW02]

> A strategy for player $\sigma \in \{Player, Environment\}$ is function $f_\sigma : V^\star V_\sigma \rightarrow V$, which determines a successor for player σ respecting the history of already visited nodes.

In the strategy, the choice of a successor may depend on the already visited nodes. If the strategy for a player σ does not depend on the history of visited nodes, the strategy is called a memoryless strategy.

Definition 13: Memoryless Strategy [GTW02]

> A memoryless strategy for player σ is a function $f_\sigma : V_\sigma \rightarrow V$, where $\forall v \in V_\sigma : f_\sigma(v) \in vE$. The choice of the successor depends only on the current node. In bipartite games one can even write: $f_\sigma : V_\sigma \rightarrow V_{\bar{\sigma}}$.

A **winning memoryless strategy** for player σ is a strategy that produces only winning plays for player σ. The winning region for a player is the set of game nodes from which he will win the game. A game is called *determined*, if the winning regions partition the set of nodes V. The following theorem is proven in [GTW02].

Theorem 1: Memoryless Determinacy of Parity Games [GTW02]

> Every parity game is determined. Whenever a player σ wins a play, he has a memoryless winning strategy.

Chapter 4

Architecture Transformations

In order to categorize the Distributed Synthesis Problem, Finkbeiner and Schewe [FS05] found a criterion to determine whether or not an instance of the Distributed Synthesis Problem belongs to the decidable subclass.

This criterion is called *Information Fork*. Additionally, they introduced architecture transformations, which transform every architecture without an Information Fork into a strictly hierarchical architecture. Distributed games can handle pipeline architectures only. In order to be able to handle all strictly hierarchical architectures, a transformation is introduced, which converts every strictly hierarchical architecture into a pipeline architecture. In general, it is impossible to transform every strictly hierarchical architecture into a pipeline, but ReaSyn assumes no communication delay. This setup renders the later defined transformation possible. Pipeline architectures can be used to construct a distributed game, which is used to find an implementation for an instance of the Distributed Synthesis Problem.

The below defined transformations all share the following property. The Distributed Synthesis Problem $(\mathcal{A}, \varphi)$ is realizable, if and only if the transformed problem $(\mathcal{A}', \varphi')$ is realizable. An implementation for the original problem can be constructed from an implementation of the transformed problem.

4.1 Information Forks

An instance of the Distributed Synthesis Problem is decidable if and only if it has no information fork [FS05]. An information fork in an architecture is a situation, where two processes receive incomparable information in a way that the information received by either process cannot be completely deduced from the other one.

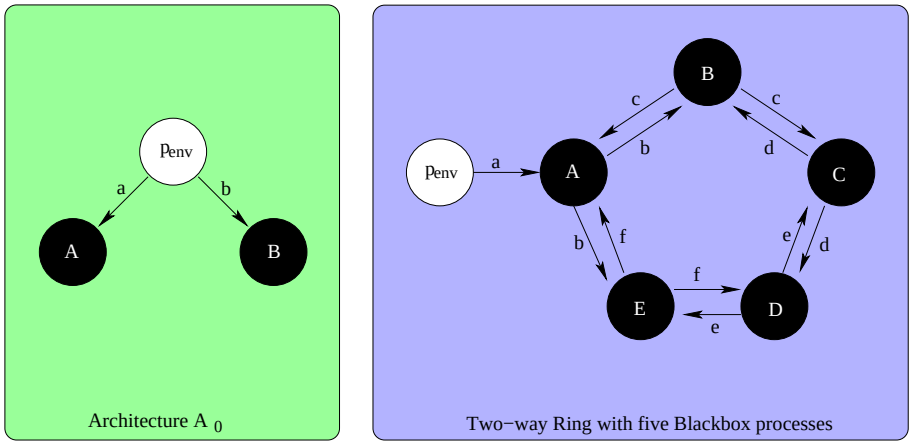


FIGURE 4.1 Architectures with Information Fork

Definition 14: Information Fork [FS05]

> An information fork is a tuple $F = (P', S', p, p')$ where P' is a subset of P, S' is a subset of the signals disjoint from $I(p) \cup I(p')$ and $p, p' \in P'$ are two different processes. F is an information fork if and only if P' together with the edges of E_S labeled with elements of S' form a subgraph, which is rooted in p_{env}, and there exist two nodes $q_1, q_2 \in P'$ that have edges to p, p', respectively, such that $O(q_1, p) \not\subseteq I(p')$ and $O(q_2, p') \not\subseteq I(p)$.

Figure 4.1 shows two architectures with an information fork. The architecture $\mathcal{A}_0$ contains the information fork $(\{p_{env}\}, \emptyset, A, B)$. In Architecture $\mathcal{A}_0$, A and B have incomparable levels of information, both directly receiving different input from the environment. However, neither process A nor B have any knowledge of the other's information. In Figure 4.1 on the right, $(\{p_{env}, A, B, E\}, \{a, b\}, C, D)$ is an information fork of the two-way ring with five processes. The environment can send information to C via a, b, and c and to D via a, b, and f. In this case, C and D have different information available, which cannot be deduced from the other process.

4.2 Finding Information Forks

[FS05] provide an algorithm to decide whether a given architecture has an information fork. It is based on the observation that every architecture without an information fork can be ordered according to the relative knowlegde of the processes. To define this order, a number of preliminary definitions are needed: Let $E_p = \{s \in S \mid s \notin I(p)\}$ be the set of edges that carry information invisible to p. U_p is defined as the set $\{q \in P \mid$ there is no path from p_{env} to q

in $(P, E_p)\}$.

Definition 15: Preorder $\preceq$ [FS05]

> Let $p, p' \in B$ then the following holds: $p \preceq p' \Leftrightarrow p' \in U_p$

The existence of an information fork in $\mathcal{A}$ is not examined on $\mathcal{A}$ itself, but on a related architecture $\mathcal{A}'$. $\mathcal{A}'$ is obtained by eliminating idle processes from A, i.e., processes $p \in P.\, O(p) \neq \emptyset$. The architecture $\mathcal{A}'$ is called:

Definition 16: $idlefree(\mathcal{A})$ [FS05]

> The related idle-free Architecture $\mathcal{A}'$ to an architecture $\mathcal{A}$ is defined as follows:
>
> - $P' = P^{-} \cup \{p_{env}\}$
> - $E'_S = E_S \cap P' \times S \times P'$
> - $S' = S$
> - $I' = I \mid_{P'}$
> - $O' = O \mid_{P'}$

Definition 17: Order [FS05]

> An architecture $\mathcal{A}$ is called ordered by a surjective function $f : P \rightarrow \mathbb{N}_n$ for some $n \in \mathbb{N}$, if $\{p_{env}\}$ is the preimage of 1 and $\forall p, p' \in P. f(p) \leq f(p') \Leftrightarrow p \preceq p'$. If f is bijective, $\mathcal{A}$ is called strictly ordered. $\mathcal{A}$ is called weakly ordered, if $idlefree(\mathcal{A})$ is ordered.

The algorithm to determine wheter or not $\mathcal{A}$ has an information fork functions as follows:

1. compute $idlefree(\mathcal{A})$
2. compute $\preceq$
3. If $\forall p_1, p_2 \in P.\, p_1 \preceq p_2 \vee p_2 \preceq p_1$, then $\mathcal{A}$ does not contain an information fork; otherwise it does.

The transformations described in this section turn every architecture that does not contain an information fork into a strictly ordered acyclic architecture. The transformations are introduced in [FS05].

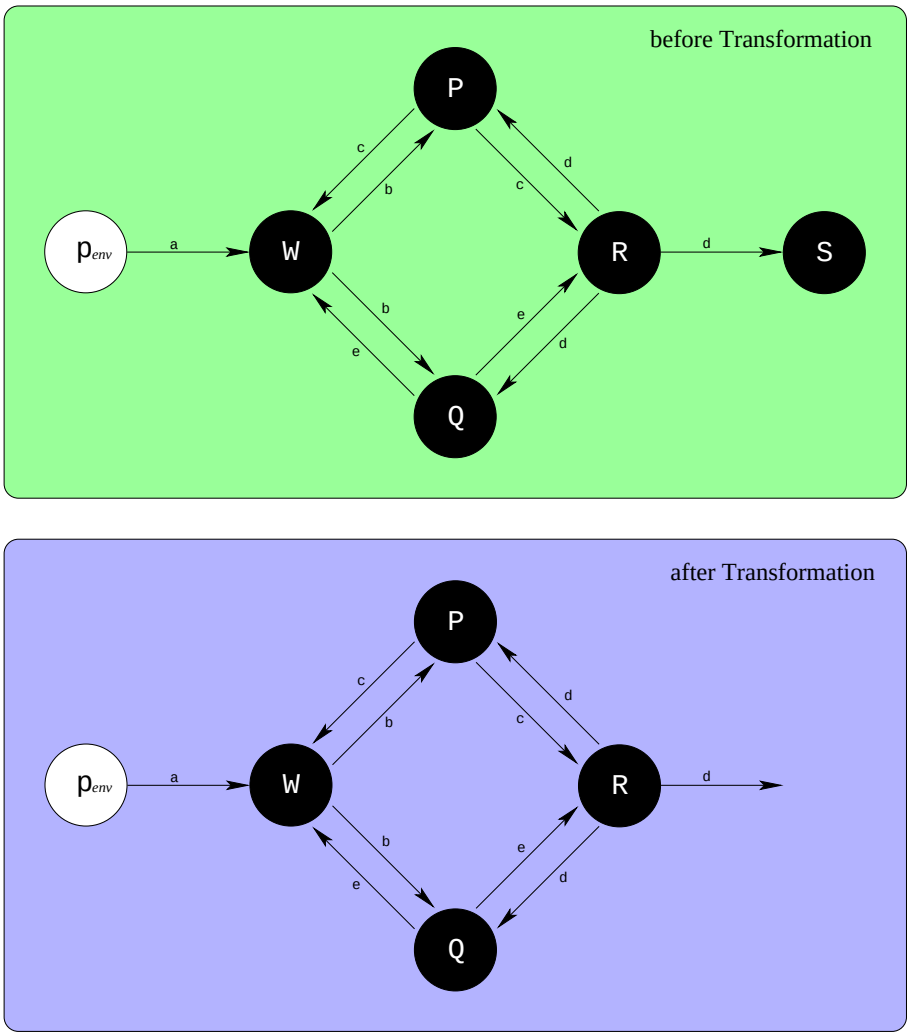


FIGURE 4.2 Elimination of Idle Processes

4.3 Elimination of Idle Processes

The definition of the related architecture without idle processes is given in Definition 16.

Figure 4.2 shows an architecture before and after the elimination of idle processes. Process S has no outgoing signals. The calculations of process S hence have no influence over the behavior of the system. Therefore, it is removed.

4.4 Quotient Architecture

In a quotient architecture, processes with the same level of information are clustered and treated as one single process. This transformation is based on the observation that equally informed processes can simulate each other. In the following the notion of "equally informed" is formally defined.

Definition 18: Equally Informed Processes [FS05]

> Two processes $p, p' \in P$ are equally informed ($p \sim p'$) if and only if $p \preceq p' \wedge p \succeq p'$.

The quotient architecture is then defined as $\mathcal{A}' = \mathcal{A}/\sim$:

Definition 19: Quotient Architecture [FS05]

> Let $g : P^- \to (P^-/\sim)$ be the function, which maps a process p to its equivalence class with respect to $\sim$. The pseudo inverse function $g^{-1} : 2^{P'} \to 2^P$ is given as $g^{-1}(\tilde{p}) \stackrel{def}{=} \{p \in P \mid g(p) \in \tilde{p}\}$. Then, the architecture $\mathcal{A}' = \mathcal{A}/\sim$ is called the quotient architecture of $\mathcal{A}$ with respect to $\sim$ if, and only if
>
> - $P' = (P^-/\sim) \cup \{p_{env}\}$
> - $E'_S = \bigcup_{(p,s,p') \in E_S} \{(g(p), s, g(p')) \mid g(p) \neq g(p')\}$
> - $S' = S$
> - $I'(p') = \bigcup_{p \in g^{-1}(p')} I(p)$
> - $O'(p') = \bigcup_{p \in g^{-1}(p')} O(p)$

Figure 4.3 shows an architecture before and after building the quotient architecture. The processes P and Q have the same information available, as they both can only receive information through signals a and b. Therefore, they are collapsed into the node $\bar{P}$, which simluates both original nodes. This implies that $\bar{P}$ receives all input sent to P and Q and produces all output, which P and Q would have produced.

4.5 Elimination of Feedback Edges

The last transformation removes all feedback edges. Feedback edges are edges leading from a process p to a better informed process p', in other words $p' \preceq p$. These edges can be removed, since the better informed process can predict the

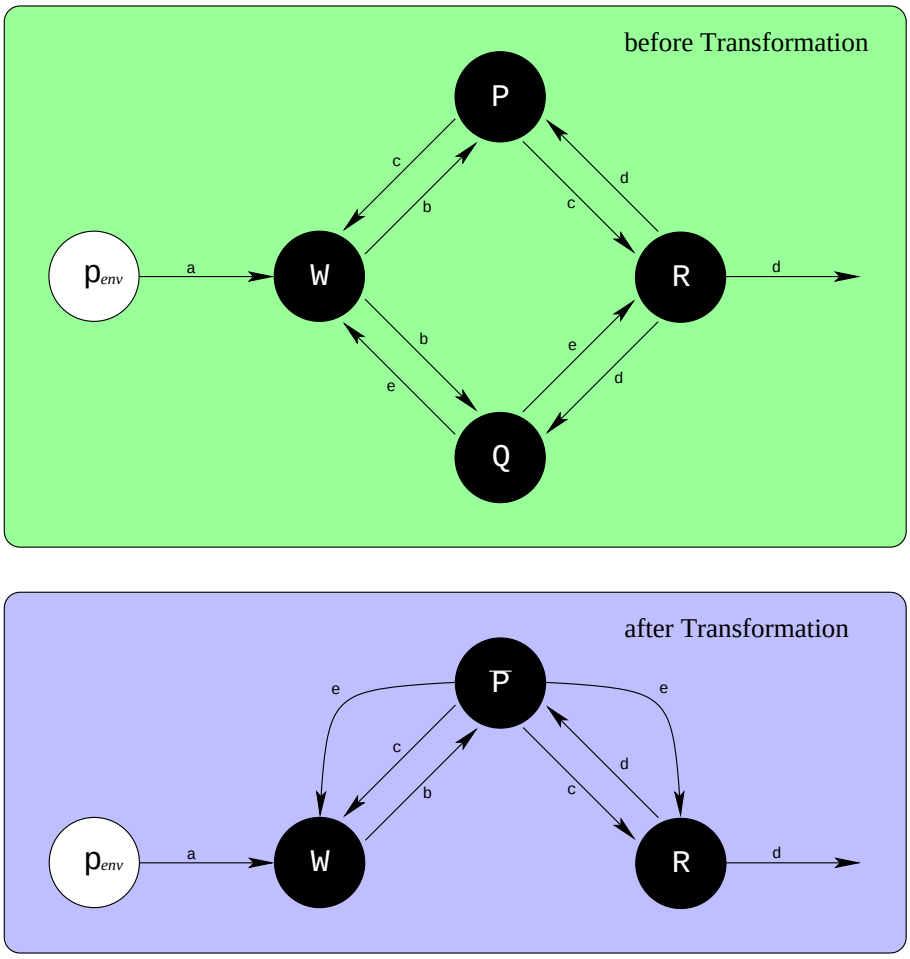


FIGURE 4.3 Quotient Architecture

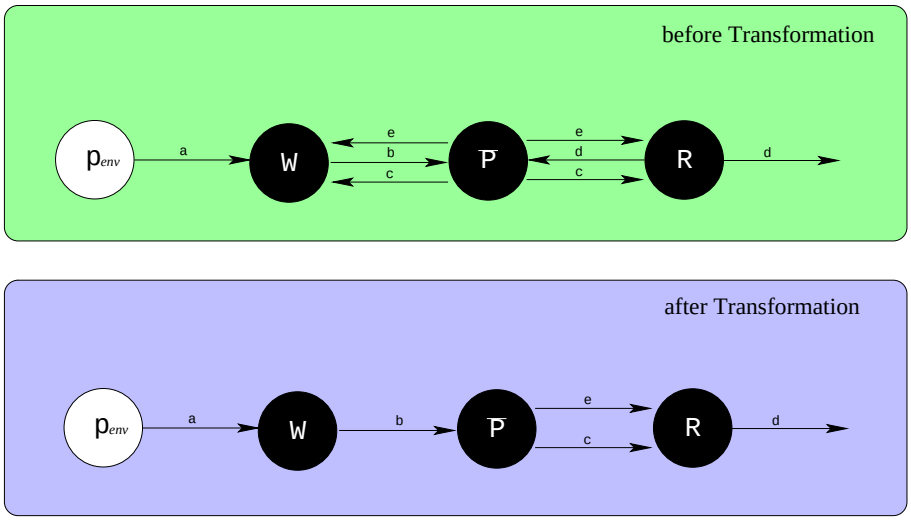


FIGURE 4.4 Elimination of Feedback Edges

feedback from the less informed process. Formally, the related architecture without feedback edges $\mathcal{A}' = acycle(\mathcal{A})$ is defined as:

Definition 20: $acycle(\mathcal{A})$ [FS05]

Let $\mathcal{A}$ be a strictly ordered architecture. Then $acylce(\mathcal{A}) \stackrel{def}{=}$

- $P' = P$
- $S' = S$
- $E'_{S'} = \{(p, s, p') \in E \mid p \preceq p'\}$
- $I'(p) = \{s \in I(p) \mid \forall q \in WRITE(s) : q \preceq p\}$
- $O' = O$

Figure 4.4 shows the elimination of feedback edges. Before the transformation, the processes are ordered according to their level of information (decreasing from left to right). Thus, having strictly more information, process $\bar{P}$ can foresee the output of R. The feedback edge from R to $\bar{P}$ is hence superfluous and is removed.

4.6 Pipeline Encoding

In addition to the transformations proposed by [FS05], ReaSyn transforms the resulting strictly hierarchical acyclic architecture into a pipeline architecture. The transformation is necessary to enable ReaSyn to handle all strictly hierarchical architectures. In order to meet the assumptions from [Mau05, MW03],

who expect pipeline architectures, ReaSyn aligns all processes of a strictly hierarchical architecture into a pipeline according to their relative knowledge using the relation $\preceq$. They are consecutively numbered according to their position in the pipeline. ReaSyn then adds new signals to pass information through the pipeline to less informed processes. Additionally the specification is modified to ensure that the values of the added signals equal the original ones. As mentioned before, the transformation is possible, because ReaSynassumes no communication delay. In an asynchronous setting, this transformation would yield two unequal architectures. Formally, the pipeline architecture $\mathcal{A}' = pipe(\mathcal{A})$ is defined by:

Definition 21: $pipe(\mathcal{A})$

The number $n_p = |\{q \in P \mid q \preceq p\}|$ is the number of processes, which are better informed than p. At the same time, it is the position of the process in the pipeline. For a signal $s \in S$ let s_I be defined as $s_I = \{s_i \mid s \in S \wedge \min\{n_p \mid p \in WRITE(s)\} < i \leq \max\{n_p \mid p \in READ(s)\}\}$.

Then $pipe(\mathcal{A}) \stackrel{def}{=}$

- $P' = P$
- $S' = S \cup \bigcup_{s \in S} s_I$
- $E'_{S'} = \{(p, s, q) \in E_S \mid n_p + 1 = n_q\}$
  $\cup \{(p, s_{(n_p - n_r)}, q) \mid r \in WRITE(s),\ n_p + 1 = n_q\}$
- $I'(p) = (I(p) \setminus \{s \in S \mid w \in WRITE(s) n_w + 1 \neq n_p\})$
  $\cup \bigcup_{s \in S} \{s_{(n_q - n_r)} \mid q \in P,\ r \in WRITE(s),\ n_q + 1 = n_p\}$
- $O'(p) = O(p) \cup \bigcup_{s \in S} \{s_{(n_p - n_r)} \mid r \in WRITE(s)\}$

Additionally, the specification for the architecture has to be modified:

$$\varphi' = \varphi \wedge \bigwedge_{s \in S,\, i=1}^{|s_I|} \Box(s = s_i)$$

Recall that the set $WRITE(s)$ for a signal $s \in S$ is always a singleton set. Figure 4.5 shows a strictly hierarchical acyclic architecture before and after the transformation into a pipeline architecture. Process P is better informed than process Q, hence the information from signal b has to be passed to Q in the pipeline architecture. This is achieved by adding the signal b_1 with writer P and reader Q. In order to ensure the equality of the values of b and b_1, the specification is extended to $\varphi' = \varphi \wedge \Box(\mathsf{b}{=}\mathsf{b}_1)$.

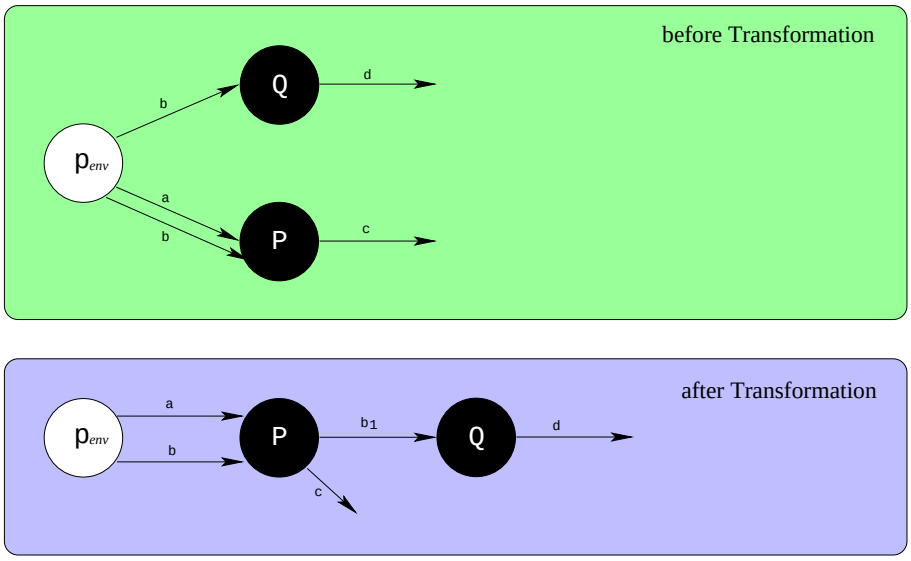


FIGURE 4.5 Pipeline Transformation of a Strictly Hierarchical Acyclic Architecture. The modified specification is $\phi = \varphi \wedge \Box(b = b_1)$

4.7 Game Encoding

The construction of the architecture games is introduced in [MW03]. Given a strictly hierarchical acyclic architecture $\mathcal{A} = (P, S, E, p_{env}, O, I)$ one obtains a bipartite two-player game arena $\mathfrak{A}_p = (V_{Player}, V_{Env}, E)$ for a process $p \in P$ through the following construction:

- $V_{Player} = D(I(p))$
- $V_{Env} = (D(I(p)) \to D(O(p)))$ that is the set of functions from $D(I(p))$ to $D(O(p))$.
- $E = \{(v, v'), (v', v) \mid v \in V_{Player}, v' \in V_{Env}\}$

This encoding provides a bipartite two-player game, in which the Player chooses the behavior of the process and the Environment chooses the input of the process. A game obtained by the above transformation simulates all possible behaviors of a process and has thus no winning condition.

Chapter 5

Automata

Definition 22: Automata

> A non-deterministic word automaton A is given through a tuple $A = (\Sigma, V, \delta, v_0, ACC)$. Σ is the alphabet of the automaton, V is a finite set of nodes. $\delta \in V \times \Sigma \to 2^V$ is the transition function. v_0 is a designated initial node and ACC a winning condition as in Definition 10.

In analogy to games, automata with Büchi (Rabin, parity) winning conditions are called Büchi (Rabin, parity) automata. An automaton is called deterministic, if $\forall q \in Q, \sigma \in \Sigma : |\delta(q, \sigma)| \leq 1$. In addition to the classic definition of automata, a very weak alternating automaton is defined. Alternating automata are introduced by Müller and Schupp [MS84, MS87, MS95]. Rhode [Rho97] introduced the very weak alternating automata to handle transfinite words. However, the slightly modified definition of Gastin and Oddoux [GO01] is used:

Definition 23: Very Weak Alternating Automata [GO01]

> A five tuple $A = (\Sigma, V, \delta, v_0, ACC)$ is called a very weak alternating automaton, if and only if Σ and V are defined as above $\delta \in Q \to 2^{\Sigma \times Q'}$, where Q' denotes the set of conjunctions of elements of Q. The empty conjunction is denoted by $\top$. Additionally, $v_0 \in Q'$ is the initial node and $ACC \subseteq Q$ is a winning condition as in Definition 10. Finally, there exists a partial order $\prec$ on Q such that $\forall q \in Q, \sigma \in \Sigma : \forall q' \in \delta(\sigma, q) : q \prec q'$.

In order to adapt the definition of winning conditions for automata, a "play of a game" in Definition 10 has to be replaced by a run of an automaton. Using

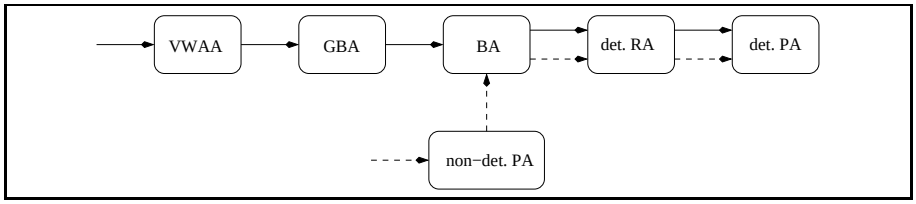


FIGURE 5.1 The two transformation chains. The first begins with a very weak alternating co-Büchi automaton and follows the solid arrows; the second starts with a non-deterministic parity automaton and follows the dashed arrows.

the following definition of a run of an automaton instead of a play, the winning conditions from Definition 10 apply to automata as well.

Definition 24: Run

A run π of a (non-)deterministic word automaton A is defined as follows:

- $\pi = v_0^{\sigma_0} v_1^{\sigma_1} v_2^{\sigma_2} \ldots$ (infinite run)
- $\pi = v_0^{\sigma_0} \ldots v_n$ (finite run)

where $v_i \in V$ and $\sigma_i \in \Sigma$. Furthermore, π has the following property: $v_{i+1} \in \delta(v_i, \sigma_i)$.

A run of a very weak alternating automaton on a tree $\langle T, l \rangle$ is a tree $\langle T_r, r \rangle$, where the root is labeled with the initial node of the automaton and every other node is labeled by an element of $\Upsilon^\star \times V$. Each node v in T_r corresponds to a node in T and v is labeled by (x, q) describing that a copy of the automaton is reading a node x from T in state q.

A non-deterministic automaton $\mathcal{A}$ accepts a sequence $s = \sigma_1 \sigma_2 \ldots$, if and only if there exists an accpeting run $\pi = v_1^{\sigma_1} v_2^{\sigma_2} \ldots$ of $\mathcal{A}$. A run $\langle T_r, r \rangle$ of a very weak alternating automaton is accepting, if all its infinite paths satisfy the acceptance condition.

ReaSyn uses the below mentioned transformations on two occasions. First, the specification of the Distributed Synthesis Problem is transformed into a very weak alternating co-Büchi automaton. Subsequently, it is transformed into a generalized Büchi automaton, a Büchi automaton, a deterministic Rabin automaton, and finally a deterministic Parity automaton. The second transformation chains starts with a non-deterministic Parity automaton which is converted into an equivalent Büchi automaton. Subsequently, the Büchi automaton is transformed into a deterministic parity automaton as before. Figure 5.1 shows the two transformation chains used by ReaSyn. The chain following the solid arrows is used to convert the specification into a parity game,

while the second is deployed to transform games with the "all-path" winning condition into a parity games.

5.1 Very Weak Alternating Automaton to Generalized Büchi Automata

The first transformation converts a very weak alternating co-Büchi automaton into a generalized Büchi automaton. This transformation is nescessary, because the Büchi automaton, which is obtained from a direct transformation from the very weak alternating automaton, is usually significantly too large [GO01]. A prior transformation to a generalized Büchi automaton reduces this growth of theautomaton. A generalized Büchi automaton is a Büchi automaton with an acceptance condition on transitions instead of nodes. The generalized Büchi automaton $GBA_\varphi = (\Sigma, V, \delta, v_0, ACC)$ is obtained from the very weak alternating automaton $A_\varphi = (\Sigma', V', \delta', v_0', ACC)$ by the following transformation:

- $\Sigma = \Sigma'$
- $V = 2^{V'}$ (the conjunctions of nodes of V')
- $v_0 = v_0'$
- $ACC = \{T_f \mid f \in ACC' \text{ where } T_f = \{(e, \alpha, e') \mid f \notin e' \text{ or } \exists(\beta, e'') \in \delta'(f) : \alpha \subseteq \beta \text{ and } f \notin e'' \subseteq e'\}$
- $\tilde{\delta}(q_1 \wedge \cdots \wedge q_n) \stackrel{def}{=} \bigotimes_{i=1}^{n} q_i$
- δ is the set of $\preceq$-minimal transitions of $\tilde{\delta}$ where $\preceq$ is defined by: $t' \preceq t \Leftrightarrow t = (e, \alpha, e'), t' = (e, \alpha', e''), \alpha \subseteq \alpha', e'' \subseteq e' \text{ and } \forall T \in ACC : t \in T \Rightarrow t' \in T$.

5.2 Generalized Büchi Automata to Büchi Automata

The transformation from a generalized Büchi automaton to a Büchi automaton is well known [GO01]. The Büchi automaton $B_\varphi = (\Sigma, V, \delta, v_0, ACC)$ is obtained as the result of the transformation of the generalized Büchi automaton $GBA_\varphi = (\Sigma', V', \delta', v_0', ACC')$ with $ACC = \{T_1, \ldots, T_r\}$ as follows:

- $\Sigma = \Sigma'$
- $V = V' \times \{0, \ldots, r\}$

- $v_0 = (v'_0, 0)$
- $ACC = V' \times \{r\}$
- $\delta((q,j),\alpha) = \{(q',j') \mid q' \in \delta'(q,\alpha) \text{ and } j' = next(j,(q,\alpha,q')\}$

  where $next(j,t) \stackrel{def}{=} \begin{cases} \max\{j \leq i \leq r \mid \forall j < k \leq i : t \in T_k\} & \text{if } j \neq r \\ \max\{0 \leq i \leq r \mid \forall 0 < k \leq i : t \in T_k\} & \text{if } j = r \end{cases}$

Theorem 2: Transformation Correctness

1. $\mathcal{L}(A_\varphi) = \mathcal{L}(GBA_\varphi)$
2. $\mathcal{L}(GBA_\varphi) = \mathcal{L}(B_\varphi)$

A proof of the theorem can be found in [GO01].

5.3 Non-deterministic Büchi Automaton to Deterministic Rabin Automaton

The determinization of a non-deterministic Büchi automaton results in a deterministic Rabin automaton and was introduced by Safra [Saf88]. The algorithm presented here is taken from [THB95]. A proof of correctness can be found in [Saf88]. The complexity of this procedure is at least $2^{O(n \log n)}$, where n is the number of nodes of the non-deterministic Büchi automaton.

A node in the Rabin automaton represents the nodes of the Büchi automaton that can occur on a non-deterministic run. Therefore, each node of the Rabin automaton is a labeled ordered tree with the following properties:

i: Every node of the tree has a color. It is either white or green.

ii: There exits a complete ordering on the children of each node. The ordering resembles the age of the children. (i.e., if a and b are children of the same node, either a is older than b or vice versa).

iii: The label of each node of the tree is a subset of the nodes of the Büchi Automaton with the following restrictions: First, the union of the labels of the children is a proper subset of the label of the parent node. Second, two nodes neither of which is an ancestor of the other must have disjoint sets as their labels.

iv: Each node has a distinct name, which is a number between 1 and twice the number of nodes of the Büchi automaton (n). This is always possible because of the previous property, where a tree has at most n nodes.

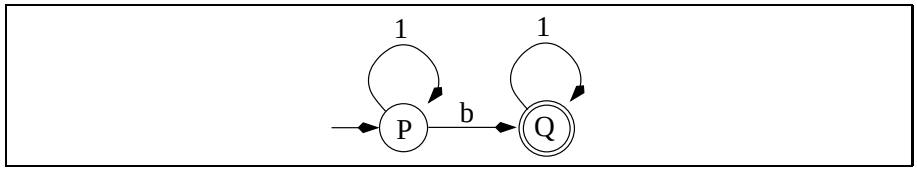


FIGURE 5.2 Büchi automaton for the formula $\Diamond b$ generated by LTL2BA[GO01]

A node a in a labeled ordered tree is said to be *right* to a node b, if either a is older than b or an ancestor of a is an older sibling of an ancestor of b.

For a given non-deterministic Büchi automaton $\mathcal{B} = (\Sigma, V, \delta, v_0, F)$ the deterministic Rabin automaton $\mathcal{R} = (\Sigma, V', \delta', v_0', C)$ is obtained from the following inductive algorithm [THB95]:

- v_0' is a node, which contains a white colored single node tree. The label is the set $\{v_0\}$ containing the initial node of the Büchi automaton. The name of the node is set to 1.
- Given a node v and an element $\sigma \in \Sigma$, the successor $v' = \delta(v, \sigma)$ is obtained as follows:
  1. The tree of v' is copied from the tree of v.
  2. set the color of all nodes in the tree of v' to white.
  3. For every node in the tree of v' replace its label L with $\delta(L, \sigma)$.
  4. For every node in the tree of v', if its label L contains any accepting nodes from F, create a new "youngest" child labeled with $F \cap L$. (all final nodes in the label L.)

     the tree obtained by steps 2-4 may violate the above assumptions (i - iv) that are made on the tree. Therefore, the remaining steps restore the premises.
  5. For every node $q \in V$ appearing in the label of a node t of the tree, remove q from every label of nodes *right* to t.
  6. Remove all nodes with empty labels.
  7. For every node t, whose label L is equal to the union of the label of its children, remove all descendants of t and set its color to green.
  8. Give every newly created node in the tree a new name between 1 and $2*n = 2*|V|$, which was not used in the tree. All other nodes keep their names.

Beginning with the initial node q_0, it is now possible to inductively define the set of nodes V' and the transition function δ, respectively, of the deterministic Rabin automaton. Following this procedure, one only computes nodes reachable from the initial node. In order to define the acceptance pairs of

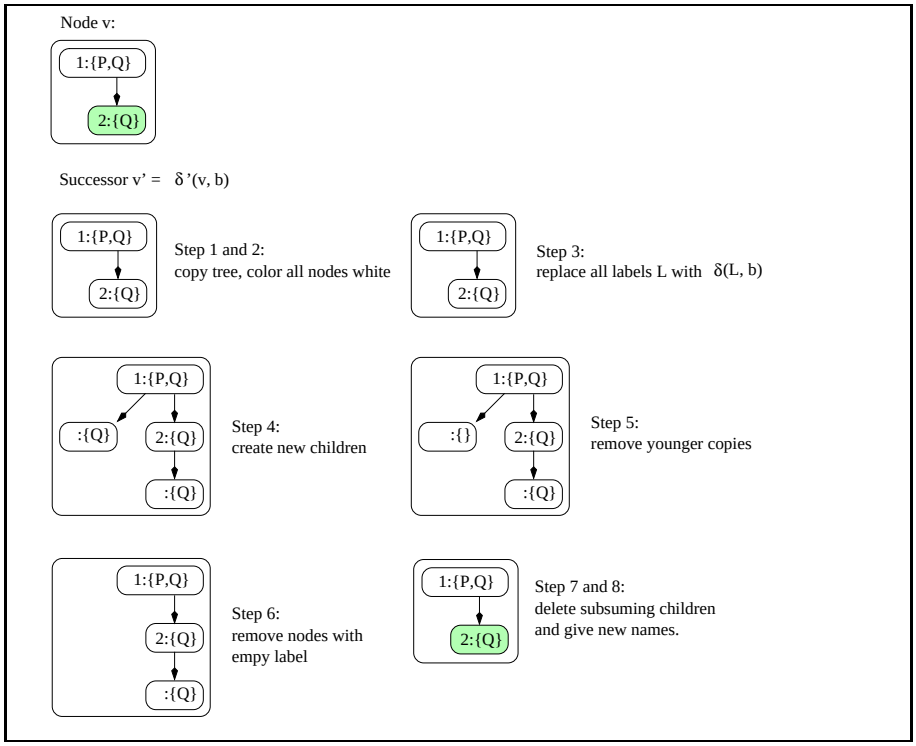


FIGURE 5.3 Successor of a node v in the Determinization of the Automaton from Figure 5.2

the Rabin automata, let $G_i \subseteq V'$ be the set of nodes, whose trees have a green node named i. Additionally, let $R_i \subseteq V'$ be the set of nodes, whose trees do not have a node named i. The acceptance pairs are then defined as $ACC = \{(G_i, R_i) \mid 1 \leq i \leq 2n\}$.

A pair $(\emptyset, R_i)$ is irrelevant, because it will never accept a run of the automaton and can therefore be removed from C. This is an important simplification, since the following transformation into a parity automaton depends exponentially ($O(n!)$) in the number n of acceptance pairs in the Rabin automaton. Additionally, all pairs $(G_i, \emptyset)_{i \in I}$ can be combined into a single pair $(\bigcup_{i \in I} G_i, \emptyset)$.

Figure 5.3 illustrates the eight steps of the presented algorithm. It computes the successor of the node v with $\sigma = b$. The underlying Büchi automaton is shown in Figure 5.2. In the first two steps, the tree is simply copied and colored white. Next, the labels L are replaced with the set of successors $\delta(L, b)$. Hence, $\{Q\}$ is replaced with $\{Q\}$ and $\{P, Q\}$ with $\{Q, P\}$. This results in the same tree. In step four, new children are created for every final node in the label of a node. These new nodes do not have a name. If necessary, they will receive names in the last step after restoring the properties that must hold for every tree. Step five then removes all nodes from labels that already appear *right* of the processed node. In this case, Q is removed from the label of the new child of node 1. Step six causes the new child of node 1 to be deleted, while in the

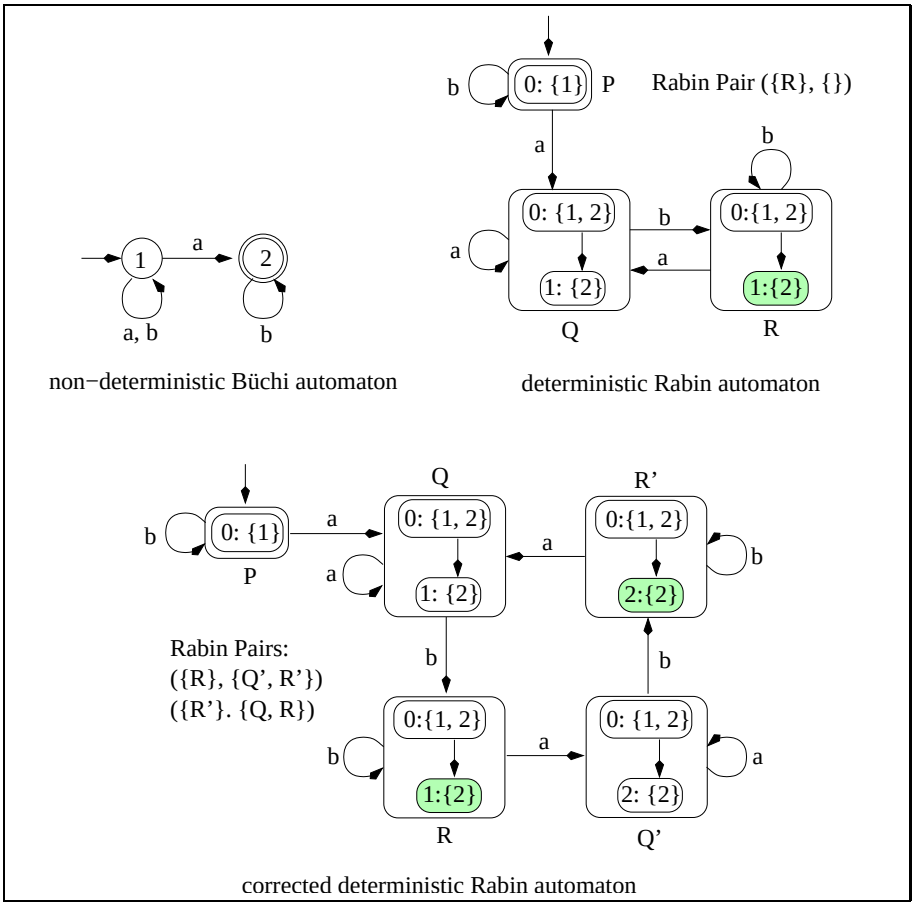


FIGURE 5.4 Counter example for Safra algorithm with only n tree node names. Green tree nodes are shaded. Nodes 1 and P are initial nodes.

next step, the descendants of node 2 are removed. Node 2 becomes green. In the final step, new nodes, which do not have a name, get a new name. As there are no new nodes left, nothing is to be done.

Remark: Our algorithm uses at most $2n$ tree node names, where n is the number of nodes of the non-deterministic Büchi automaton. The algorithm presented in [THB95] uses only n tree node names. This may lead to a situation where a name for a deleted tree node is reused directly. Figure 5.4 shows such an example. Observe, that the languages of the automata are different as the deterministic Rabin automaton accepts the word $(ab)^\omega$, which the Büchi automaton does not. However, dropping the limitation to n tree node names corrects the algorithm.

5.4 Rabin Automaton to Parity Automaton

The translation of the deterministic Rabin automaton into a deterministic parity automaton is almost the same as the construction given in [FS05] for Streett to parity automata. Due to the complementarity of Streett and Rabin automata only the definition of the coloring function has to be complemented. This leads to the following construction:

Given a deterministic Rabin automaton $\mathcal{R} = (\Sigma, V, \delta, v_0, (G_i, R_i)_{i \in I})$, one can construct an equivalent deterministic max parity automaton $\mathcal{P} = (\Sigma, V', \delta', v_0', \alpha)$ as follows:

- $V' = V \times \mathrm{perm}(I) \times I \times I$ where $\mathrm{perm}(i)$ denotes the set of permutations of I
- $v_0' = (v_0, id_I, 0, 0)$
- $(q', \pi', r', g') \in \delta'((q, \pi, r, g), \sigma)$ iff
  - $(q') \in \delta(q, \sigma)$,
  - $\pi' = (p_1, p_2, \ldots, p_k)$ is obtained from $\pi = (j_1, j_2, \ldots, j_k)$ by shifting all numbers j_l with $q_v \in R_{j_l}$ to the left,
  - r' is the greatest number such that $q \in \mathbf{R}_{j_{r'}}$ (or 0 if q is in no red set) and
  - g' is the greatest number such that $q \in \mathbf{G}_{j_{g'}}$ (or 0 if q is in no green set)
- $\alpha : (q, \pi, r, g) \mapsto 2g$ if and only if $g > r$ and
- $\alpha : (q, \pi, r, g) \mapsto (2r) + 1$ otherwise

The correctness of this transformation follows from the correctness of the construction in [FS05] taking into account that the winning condition is "inverted". The min parity automaton needed for the game encoding is obtained by a simple parity inversion of the automaton nodes:

Let m be the maximal parity occuring in the deterministic max parity automaton $\mathcal{P} = (\Sigma, V, \delta, v_0, \alpha)$. An equivalent min parity automaton $\mathcal{P}' = (\Sigma, V, \delta, v_0, \alpha')$ is obtained by the following definition:

$$\alpha'(v) \stackrel{def}{=} \lceil m \rceil - \alpha(v)$$

where $\lceil m \rceil$ is the smallest even natural number that is greater than or equal to m.

5.5 Parity Automaton to Büchi Automaton

The transformation described in this section is used to convert an "all-path parity condition", which is a ω-regular winning condition, into a parity condition. The transformation is described in detail in Chapter 7.

The non-deterministic min parity automaton $\mathcal{P} = (\Sigma, V, \delta, v_0, \alpha)$ can be converted into an equivalent non-deterministic Büchi automaton $\mathcal{B} = (\Sigma, V', \delta', v_0, F)$ by the following transformation:

Let $S = \{\alpha(v) \mid v \in V\} \cap 2 \cdot \mathbb{N}$ be the set of the even parities appearing in $\mathcal{P}$

- $V' = V \cup \bigcup_{i \in S} \{(v, i, j) \mid v \in V, \alpha(v) = j, i \leq j\}$
- $\delta'(v, \sigma) = \begin{cases} \delta(v, \sigma) \cup \\ \quad \{(v', i, i) \mid v' \in \delta(v, \sigma) \\ \qquad \wedge\, \alpha(v') = i \\ \qquad \wedge\, i \text{ is even}\} & , v \in V \\ \\ \{(w, i, j) \mid \alpha(w) = j \wedge i \leq j \\ \qquad \wedge\, w \in \delta(v', \sigma)\} & , v = (v', i, j) \in V' \setminus V \end{cases}$
- $F = \{(v, i, i) \mid v \in V\}$

Theorem 3:

$$\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{B})$$

Proof:

"$\subseteq$" Let $s = \sigma_1 \sigma_2 \cdots \in \Sigma^\omega$ be a word in $\mathcal{L}(\mathcal{P})$ and $\pi = v_1 v_2 \ldots$ the run of $\mathcal{P}$, which accepts s. Additionally, let p_{min} be the minimal parity from the infinity set $\mathrm{Inf}(\chi(\pi))$. p_{min} is even, because the run π is an accepting run. Then, there is a $k \in \mathbb{N}$, such that $\alpha(v_k) = p_{min} \wedge \forall l \geq k : \alpha(l) \geq p_{min}$. The run $\pi' = v_1 v_2 \ldots v_{k-1}(v_k, p_{min}, p_{min})(v_{k+1}, p_{min}, \alpha(v_{k+1})) \ldots, v_i \in \pi$ is a run of the Büchi automaton $\mathcal{B}$. Since p_{min} occurs infinitly often as parity of a node in π some nodes (v_i, p_{min}, p_{min}) occur infinitly often in π'. Hence, π' is a accepting run of $\mathcal{B}$ and $s \in \mathcal{L}(\mathcal{B})$.

"$\supseteq$" Let $s \in \Sigma^\omega$ be a word in $\mathcal{L}(\mathcal{B})$. Therefore, there is a run $\pi = v_1 \ldots v_k(v_{k+1}, m, m)(v_{k+2}, m, n) \ldots$ of $\mathcal{B}$, which accepts s meaning, there is a node $(v_i, m, m) \in \mathrm{Inf}(\pi)$. The third component equals the parity of node v_i interpreted as node of the parity automaton. Additionally, the third component is always

greater than or equal to the second component. The sequence $\pi' = v_1 \dots v_k v_{k+1} v_{k+2} \dots$ thus states a run of the parity automaton $\mathcal{P}$, where $m = \min\{\mathrm{Inf}(\chi(\pi'))\}$ is even. Hence, π' is an accepting run and $s \in \mathcal{L}(\mathcal{P})$. □

5.6 State Space Reduction of Büchi Automata

As proposed by Etessami, Wilke, and Schuller [EWS01] the state space of Büchi automata is reduced by delayed simulation. It yields better results than direct simulation in the state space reduction. The state space reduction is achieved by computing an equivalence relation $\sim$ on the states of the Büchi automaton and subsequently building the quotient automaton with respect to $\sim$. The equivalence relation $\sim$ is computed by a two-player parity game of the players Spoiler and Duplicator. In order to check whether a node v' can simulate a node v the Spoiler places a red pebble on v, while the Duplicator places a blue pebble on v'. The Spoiler then chooses a successor of v and moves the red pebble to it using a transition labelled with a. The Duplicator has to copy this move beginning from v'. If he cannot duplicate the move from Spoiler, v' cannot simulate v. In case of an infinite play the parity condition encodes whether the Duplicator can simulate an accepting run presented by Spoiler. Given a Büchi automaton $\mathcal{B} = (\Sigma, Q, \delta, v_0, F)$, the parity game $\mathcal{G}_{delayed} = ((V_{duplicator}, V_{spoiler}, E), \chi)$ determing the delayed simulation relation $\sim$ is defined by:

- $V_{duplicator} = \{(b, q, q', a) \mid q, q' \in Q \wedge a \in \Sigma \wedge b \in \{0,1\} \wedge \exists q'' \in Q : q \in \delta(q'', a)\}$
- $V_{spoiler} = \{(b, q, q') \mid q, q' \in Q \wedge b \in \{0,1\} \wedge (q' \in F \rightarrow b = 0)\}$
- $E = \{((b, q_1, q_1', a), (b, q_1, q_2') \mid q_2' \in \delta(q_1', a) \wedge q_2' \notin F\}$
  $\cup \{((b, q_1, q_1', a), (0, q_1, q_2')) \mid q_2' \in \delta(q_1', a) \wedge q_2' \in F\}$
  $\cup \{((b, q_1, q_1'), (b, q_2, q_1', a) \mid q_2 \in \delta(q_1, a) \wedge q_2 \notin F\}$
  $\cup \{((b, q_1, q_1'), (1, q_2, q_1', a)) \mid q_2 \in \delta(q_1, a) \wedge q_2 \in F\}$
- $\chi(v) = \begin{cases} b & , v = (b, q, q') \\ 2 & , v = (b, q, q', a) \end{cases}$

A Spoiler node has parity 1 to signify that the red pebble encountered an unmatched accepting states. Two nodes $q', q \in Q$ can simulate each other, if and only if Duplicator has a winning strategy from (b, q, q') and (b, q', q) in $\mathcal{G}_{delayed}$. Let $W_{duplicator}$ be the winning region of Duplicator in $\mathcal{G}_{delayed}$. It can be determined by the algorithm presented in [GTW02, Mau05]. Hence,

$$q \sim q' \Leftrightarrow ((b, q, q') \in W_{duplicator} \wedge (b, q', q) \in W_{duplicator})$$

for an arbitrary $b \in \{0, 1\}$.

Definition 25: Quotient Automaton [EWS01]

> For a Büchi automaton $\mathcal{B} = (\Sigma, Q, \delta, v_0, F)$ and an equivalence relation $\sim$ the quotient automaton is defined as:
>
> $$\mathcal{B}/\sim = (\Sigma, Q/\sim, \delta_\sim, [v_0], F/\sim),$$
>
> where $[q]$ represents the equivalence class of q with respect to $\sim$ and $\delta_\sim$ is defined by $\delta_\sim([q], a) = \{[q'] \mid \exists(q_0 \in [q] \wedge q_0' \in [q']) : q_0' \in \delta(q_0, a)\}$

Theorem 4: Language Equality [EWS01]

> For any Büchi automaton $\mathcal{B}$: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}/\sim)$

The theorem allows to reduce the state space of any Büchi automaton using the quotient automaton with respect to delayed simulation.

Chapter 6

Specification Transformation

In order to transform an LTL formula into a two player parity game several intermediate steps are needed. First, the formula is converted into a very weak alternating co-Büchi automaton. The language of the automaton is empty, if and only if the LTL formula is not satisfiable. Subsequently the very weak alternating automaton is first commuted into a generalized Büchi automaton (Section 5.1). It is followed by the transformation into a Büchi automaton (Section 5.2), a state space reduction of the Büchi automaton (Section 5.6), the determinization of the Büchi automaton (Section 5.3), and the conversion of the resulting deterministic Rabin automaton into a deterministic minimal parity automaton (Section 5.4). Finally, the deterministic parity automaton can be encoded as a minimal parity two-player game. The following sections describe the transformation of an LTL formula into a very weak alternating automaton and the game encoding for deterministic automata.

6.1 LTL to Very Weak Alternating Automaton

The transformation of an LTL formula over a set $atom$ of atomic propositions is introduced by Gastin and Oddoux [GO01]. The size of the resulting automaton is at most the size of the LTL formula. The following abbreviations are used in the transformation:

Definition 26:

$$J_1 \otimes J_2 \stackrel{def}{=} \{(a_1 \cap a_2, e_1 \wedge e_2) \mid (a_i, e_i) \in J_i, J_1, J_2 \in 2^{\Sigma, Q'}\}$$

$$\overline{\psi} \stackrel{def}{=} \begin{cases} \{\psi\} & , \psi \text{ is a temporal formula} \\ \{e_1 \wedge e_2 \mid e_1 \in \overline{\psi_1}, e_2 \in \overline{\psi_2}\} & , \psi = \psi_1 \wedge \psi_2 \\ \overline{\psi_1} \cup \overline{\psi_2} & , \psi = \psi_1 \vee \psi_2 \end{cases}$$

For an LTL formula φ over a set $atom$ of atomic propositions the equivalent very weak alternating co-Büchi automaton (VWAA) $\mathcal{A}_\varphi = (\Sigma, V, \delta, v_0, ACC)$ is defined by:

- $\Sigma = 2^{atom}$,
- V is the set of temporal subformulas of φ,
- $v_0 = \bar{\varphi}$,
- ACC is the set of until subformulas of φ (such as $\psi_1 \,\mathsf{U}\, \psi_2$),
- δ is defined as follows: (Δ extends δ to all subformulas of φ)

$$\delta(\varphi) \quad \stackrel{def}{=} \quad \begin{cases} \{(\Sigma, \top)\} & ,\varphi = \top \\ \{(\Sigma_p, \top)\} & ,\varphi = p \in atom, \\ & \Sigma_p = \{a \in \Sigma \mid p \in a\} \\ \{(\Sigma_{\neg p}, \top)\} & ,\varphi = \neg p, p \in atom, \\ & \Sigma_{\neg p} = \Sigma \setminus \Sigma_p \\ \{(\Sigma, e) \mid e \in \bar{\psi}\} & ,\varphi = \mathsf{X}\psi \\ \Delta(\psi_2) \cup (\Delta(\psi_1) \otimes \{(\Sigma, \psi_1 \mathsf{U} \psi_2)\}) & ,\varphi = \psi_1 \mathsf{U} \psi_2 \\ \Delta(\psi_2) \otimes (\Delta(\psi_1) \cup \{(\Sigma, \psi_1 \mathsf{U} \psi_2)\}) & ,\varphi = \psi_1 \mathsf{R} \psi_2 \end{cases}$$

$$\Delta(\psi) \quad \stackrel{def}{=} \quad \begin{cases} \delta(\psi) & ,\psi \text{ is a temporal formula} \\ \Delta(\psi_1) \cup \Delta(\psi_2) & ,\psi = \psi_1 \vee \psi_2 \\ \Delta(\psi_1) \otimes \Delta(\psi_2) & ,\psi = \psi_1 \wedge \psi_2 \end{cases}$$

Using the order "subformula of" completes the automaton to be a very weak alternating co-Büchi automaton A_φ.

Theorem 5: Transformation Correctness [GO01]

$$\mathcal{L}(A_\varphi) = \{u \in \Sigma^\omega \mid u \models \varphi\}$$

6.2 Games for Automata

Finally, the automata can be transformed into games. The corresponding parity game is used to simulate the automaton in the distributed game. The distributed game resembles the computation trees of the architecture. The task of the specification game in this setting is to ensure that every path in a computation tree satisfies the specification. The encoded game does not consider that some elements of Σ may be under control of the player in the Distributed Synthesis Problem. This issue is solved by forcing the Environment to set the signal values according to the functions chosen by the player and the signal values of the external signals.

There are two transformations. They transform a deterministic parity automaton and a deterministic Büchi automaton, likewise, into a bipartite two-player parity game.

6.2.1 Deterministic Parity Automaton

Let $\mathcal{P} = (\Sigma, V, \delta, v_0, \alpha)$ be a deterministic Parity automaton. The bipartite two-player parity game $\mathcal{G} = ((V_{player}, V_{env}, E), \chi)$ results from the following transformation:

- $V_{player} = V \times \Sigma$
- $V_{env} = V$
- $E = \{(v_e, v_p), (v_p, v'_e) \mid v_p = (v_e, \sigma) \wedge \delta(v_e, \sigma) = v'_e\}$
- $\chi(v) = \begin{cases} \alpha(v) & , v \in V_{env} \\ \alpha(v') & , v = (v', \sigma) \in V_{player} \end{cases}$

The game is an initialized game starting at node $v_0 \subseteq V_{env}$.

In $\mathcal{G}$ the Environment chooses sequences of atomic propositions, while the Player chooses a successor state in the automaton. The Player wins the game, if and only if the sequence $s = \sigma_1\sigma_2 \cdots \in \mathcal{L}(\mathcal{P})$.

6.2.2 Deterministic Büchi Automaton

Let $\mathcal{B} = (\Sigma, V, \delta, v_0.F)$ be a deterministic Büchi automaton. The corresponding game $\mathcal{G} = ((V_{player}, V_{env}, E), \chi)$ is obtained by the following transformation:

- $V_{player} = V \times \Sigma$
- $V_{env} = V$
- $E = \{(v_e, v_p), (v_p, v'_e) \mid v_p = (v_e, \sigma) \wedge \delta(v_e, \sigma) = v'_e\}$
- $\chi(v) = \begin{cases} 0 & , v \in F \\ 1 & , v \notin F \vee v \in V_{player} \end{cases}$

The game is an initialized game with the initial node v_0.

As before, the Player wins the game, if and only if the Environment chooses a sequence $s = \sigma_1\sigma_2 \cdots \in \mathcal{L}(\mathcal{B})$.

Chapter 7

Winning Condition Transformation

This chapter presents a transformation, which converts an "all-path parity" winning condition into a standard parity winning condition. The "all-path parity" winning conditions are ω-regular winning conditions and appear in ReaSyn whenever a distributed game is glued [MW03, Mau05]. Glueing a game $\mathcal{G}_{orig} = ((V_{player}, V_{env}, E), \chi)$ leads to a game $G_{glue} = ((V^g_{player}, V^g_{env}, E^g), ACC)$, where the nodes are sets of node pairs and the pair components are from the original game. For Player nodes the first pair component is a Environment node, while the second is a Player node from the original game. For Environment nodes this order is reversed. The first component refers to the predecessor of the second component in a play of the original game.

Definition 27: $threads(\vec{u})$

> Let $\vec{u} = u_1 u_2 \cdots \in (V^g_{player} \cdot V^g_{env})^\omega$ be a sequence of sets of pairs in $\mathcal{G}_{glue}$. A *thread* in $\vec{u}$ is any sequence $v_1 v_2 \cdots \in (V_{player} \cdot V_{env})^\omega$, such that $(v_i, v_{i+1}) \in u_{2i-1}$ for $i \in \mathbb{N}$. The set $threads(\vec{u})$ is defined to be the set of all threads in u.

Finally, the winning condition ACC for the glued game is defined as follows:

Definition 28: "all-path parity" Condition

> A run $\vec{u}$ of the glued game $\mathcal{G}_{glue}$ as above is accepted ($\vec{u} \in ACC$), if and only if every $t \in threads(\vec{u})$ is accepted by the parity condition χ of the original game $\mathcal{G}_{orig} = (\mathfrak{A}, \chi)$.

According to [MW03] this winning condition is a regular winning condition meaning that there is a deterministic parity automaton recognizing sequences

over $(V^g_{player} \cdot V^g_{env})$ with this property. However, they do not provide a construction of the deterministic parity automaton. The construction used by ReaSyn first constructs a non-deterministic parity automaton, which accepts such a sequence, if there is a thread that does not satisfy the parity condition. This automaton is then converted into an equivalent non-deterministic Büchi automaton (Section 5.5), which is subsequently minimized using the state space reduction from Section 5.6. Thereafter, it is transformed into a deterministic Rabin automaton (Section 5.3) and finally into a equivalent deterministic parity automaton (Section 5.4). In a last step, the parity automaton is complemented to accept all sequences that do not contain a thread, which is not accepted by the original parity condition. Finally, this parity automaton is combined with the glued game yielding a new parity game.

7.1 Non-deterministic Parity Automaton

Given a initialized parity game $(\mathcal{G}_{orig}, v_0)$ and the glued game $\mathcal{G}_{glue}$ as above. Additionally, let $V_{glue} = V^g_{player} \cup V^g_{env}$ be the set of all games nodes from the glued game. A non-deterministic mininmal parity automaton $\mathcal{P} = (\Sigma, V, \delta, v'_0, \alpha)$ accepting all sequences u over V^g, which contain at least one thread $t \in threads(\vec{u})$ not accepted by the parity condition of $\mathcal{G}_{orig}$, is constructed as follows:

- $V = V_{player} \cup V_{env}$
- $\Sigma = V_{glue}$
- $\delta(v, \sigma) = \{v' \in V \mid (v, v') \in \sigma\}$
- $v'_0 = v_0$
- $\alpha(v) = \chi(v) + 1$

7.2 Parity Game

Let $\mathcal{G} = ((V_{player}, V_{env}, E), \chi)$ be a game and $\mathcal{G}_{glue} = ((V^g_{player}, V^g_{env}, E^g), ACC)$ the corresponding glued game with the initial node v^g_0. Given a deterministic minimal parity automaton $\mathcal{P} = (\Sigma, V, \delta, v_0, \alpha)$, which accepts all plays $\vec{u}$ of $\mathcal{G}_{glue}$ that contain at least one $t \in threads(\vec{u})$, which is not accepted by χ one can construct a minimal parity game $\mathcal{G}_{eq} = ((V^{eq}_{player}, V^{eq}_{env}, E^{eq}), \chi^{eq})$ equivalent to $\mathcal{G}_{glue}$ using the following construction:

- $V^{eq}_{player} = V \times V^g_{player}$

- $V_{env}^{eq} = V \times V_{env}^{g}$
- $E^{eq} = \{((v, \sigma_1), (v', \sigma_2)) \mid (\sigma_1, \sigma_2) \in E^{g} \wedge \delta(v, \sigma_1) = \{v'\}\}$
- $\chi^{eq}((v, \sigma)) = \alpha(v) + 1$

The initial node of the game is (v_0, v_0^g), where q_0 is the initial node of the deterministic parity automaton and v_0^g is the initial node of the glued game. Note, that the parity condition is implicitly inverted to accept runs $\vec{u}$ of the glued game, where all $t \in threads(\vec{u})$ are accepted by χ.

Part II

ReaSyn

ReaSyn is a tool, which tests instances of the Distributed Synthesis Problem for realizability. It expects the user to provide an architecture and an LTL specification. Subsequently, ReaSyn checks whether the Distributed Synthesis Problem is decidable for the given architecture. In case it is decidable, ReaSyn decides wheter or not the instance of the Distributed Synthesis Problem is realizable and, in case of realizability, it generates a small finite state program.

The given architecture and the specification are first transformed into two-player parity games of Player versus Environment. They are combined into a distributed game [MW03, Mau05]. The purpose of the architecture games in the distributed game is to generate behaviors of the distributed system, while the specification game ensures that the behavior satisfies the specification. Subsequently, the distributed game is transformed into a two-player game [MW03, Mau05], which is solved yielding a winning region and a non-deterministic strategy for both players. If the winning region for player contains the initial node of the distributed game, the given instance of the Distributed Synthesis Problem is realizable and a minimized implementation is generated.

There are several optimizations to the occuring automata and games. The Büchi automata are reduced using delayed simulation. [Mau05] describes two optimizations to reduce the size of a distributed game. Additionally, during the transformation into a two player game, the parity condition of the distributed game can be approximated. The approximation can be done without growth of the distributed game, while the correct transformation described in Chapter 7 involves exponential growth of the distributed game. This approximation will not produce invalid implementations. However, ReaSyn may find some instances of the Distributed Synthesis Problem unrealizable which are in fact realizable. The correct transformation is avaiable as a commandline option. Finally, during the code generation, the generated non-deterministic strategy is examined to find a strategy that will produce a small finite state program.

Chapter 8

User Interface

ReaSyn is a program designed to solve instances of the Distributed Synthesis Problem. It accepts LTL specifications and architecture descriptions and produces an implementation for the architecture satisfying the specification. The implementation is realized as a finite state PROMELA program. ReaSyn is currently availabe as a commandline tool for LINUX systems only. The source code can be found at:

http://www.ReaSyn.org/

The architecture and specification can be provided as seperate input files. The syntax for architecture declarations is introduced in the following section. Section 8.2 gives an overview on the Syntax of the specification file. In case ReaSyn is invoked without the declaration files, it prompts the user for the declarations during runtime. The generated program is written to `stdout`. Section 8.3 describes the avaiable commandline options.

8.1 Architecture Syntax

An architecture consists of processes and signals. The processes communicate through these signals. An architecture can therefore be represented as a directed graph with processes as nodes and signals as edges. An architecture definition consists of two parts, the declaration of all processes and signals and the definition of their connections.

8.1.1 Process and Signal Declaration

First, the processes and signals are declared. As can be seen in Figure 8.1, the `Process` statement declares a new process with the given *name*. Signals

Declaration:
`Process` <*name*>;
`Signal` <*name*> <*min*> <*max*>;

Connecting Processes:
`Input` <*process_name*> <*signal_name*>;
`Output` <*process_name*> <*signal_name*>;

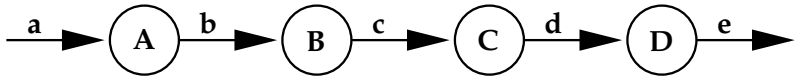
Comments:
All text between `//` and the end of the current line will be treated as comment.

FIGURE 8.1 Architecture Syntax

are declared by the `Signal` statement. In addition to their *name*, signals also have a finite integer domain of possible values. This domain is limited by *min* and *max*, respectively.

8.1.2 Connecting Processes

After all signals and processes have been declared, it is necessary to define, which process reads or writes to which signal. The `Output` statement indicates that the process *writes* values from the signal domain to the *signal*. To avoid inconsistancy, a signal can be the output signal of only one process. The number of output signals of a process, i.e. the number of signals a process writes to, is not limited. Similarly, the `Input` statement defines that the process *reads* information from the specified signal. The number of signals a process can read is not limited neither is the number of processes that read a single signal. Figure 8.2 illustrates the declaration of a four process pipeline architecture. First, the four processes `A,B,C,D` are declared using the `Process` statement. The signals `a,b,c,d,e` are declared to be binary signals. Finally, the processes are connected to the signals, e.g. '`Input A a`' defines that process `A` reads the signal `a` and '`Output C d`' indicates that process `C` writes information to signal `d`.




```
// 4 process pipeline with binary signals
Process A;
Process B;
Process C;
Process D;


Signal a 0 1;
Signal b 0 1;
Signal c 0 1;
Signal d 0 1;
Signal e 0 1;


Input A a;
Output A b;
Input B b;
Output B c;
Input C c;
Output C d;
Input D d;
Output D e;
```

FIGURE 8.2 4-Pipeline with binary signals

ARITHMETIC OPERATORS:	ARITHMETIC EXPRESSIONS:
arith_op ::= `+` | `-` | `*` | `/` | `%`	*arith* ::= <*signal name*> | <*integer value*> | *arith arith_op arith* | (*arith*)
RELATIONAL OPERATORS:	
rel_op ::= `=` | `>` | `<` | `>=` | `<=`	

FIGURE 8.3 Arithmetic Expressions

8.2 Specification Syntax

8.2.1 LTL formula

Arithmetic Expressions

Arithmetic expressions provide means to relate the specification to the architecture. Signal names (c.f. section 8.1), which can be used as integer variables and integer values are arithmetic expressions. An arithmetic expression is given as an arbitrary composition of arithmetic operators, such as `+` for addition, `-` for subtraction, `*` for multiplication, `/` for integer division, or `%` for the remainder of integer division (modulo), or of relational operators, such as `=`, `>`, `<`, `>=` and `<=`. Figure 8.3 shows a complete grammar of arithmetic expressions.

Atomic Propositions

Atomic propositions are system properties, which are either `true` or `false`. Alternatively, they can be defined as an arithmetic expression (see Figure 8.3). Any non-zero value of an arithmetic expression is interpreted as `true` while zero is regarded as `false`. Furthermore, it is possible to compose an atomic proposition of one or two atomic propositions, respectively, and any of the boolean connectives. An overview of atomic propositions is given in Figure 8.4.

Temporal Formulas

Temporal formulas define temporal relationships between system properties. They are composed of boolean expressions and any of the following temporal connectives: `[]` (always), `<>` (eventually), `U` (until), `R` (release) and `X` (next). Additionally, boolean connectives can be used to combine temporal formulas.

BOOLEAN OPERATORS:

bool_op ::= `->`
| `<->`
| `&&`
| `||`

u_bool_op ::= `!`

ATOMIC PROPOSITIONS:

atom ::= `true` | `false`
| "*arith*" | "*arith rel_op arith*"

BOOLEAN EXPRESSIONS:

bool_exp ::= *atom*
| *bool_exp bool_op bool_exp*
| *u_bool_op bool_exp*
| (*bool_exp*)

FIGURE 8.4 Boolean Expressions

TEMPORAL OPERATORS:

b_t_op ::= `U` (until)

u_t_op ::= `[]` (always)
| `<>` (eventually)
| `X` (next)

TEMPORAL FORMULAS:

formula ::= *bool_exp*
| (*formula*)
| *u_t_op formula*
| *u_bool_op formula*
| *formula b_t_op formula*
| *formula bool_op formula*

FIGURE 8.5 Temporal Formulas

Figure 8.5 contains a complete overview of the syntax of temporal formulas.

8.2.2 Büchi Automata

Büchi automata are another possibility to pass the specification to ReaSyn. The user may choose to pass a Büchi automaton to ReaSyn instead of an LTL formula. In case the Büchi automaton is deterministic, the transformation into a parity game is much faster than passing an LTL formula to ReaSyn.

In the following, the syntax for the *automaton file* is defined. A complete grammar is given in Figure 8.6. A Büchi automaton is described by a sequence of *state* declarations followed by the definition of the alphabet Σ (*sigma*) of the automaton. The states are declared using the `State` statement. `Sigma` is used to define the alphabet of the automaton. The initial state of the automaton *init* follows. It is defined using the `Init` statement. Subsequently the set *accept*, of accepting states is defined (`Accept`). Finally, the definition of the transition function δ (*delta*) is given. The `Edge` statement defines a transition from id_1

id ::= $[0-9a-zA-Z\_]^+$

id_list ::= { *id* , *id* }

state ::= `State` *id* ; [*state*]
| `State` *id_list* ;

sigma ::= `Sigma` *id_list* ;

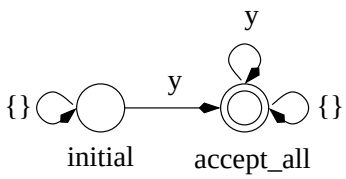
init ::= `Init` *id* ;

accept ::= `Accept` *id* ; [*accept*]
| `Accept` *id_list* ;

delta ::= `Edge` $id_1 id_2 id_3$; [*delta*]
| `Edge` $id_1 id\_list id_3$; [*delta*]

automaton ::= *state sigma init accept delta*

FIGURE 8.6 Syntax for Büchi Automata




```
State initial;
State accept_all;
Sigma y;
Init initial;
Accept accept_all;
Edge initial y accept_all;
Edge initial {} initial;
Edge accept_all {y} accept_all;
Edge accept_all {} accept_all;
```

FIGURE 8.7 Declaration of a Deterministic Büchi Automaton for the Formula $\Diamond y$

to id_3. id_2 and id_list, respectively, represent the label of the edge. Figure 8.7 shows the declaration of a deterministic Büchi automaton for the formula $\Diamond y$.

8.3 Options

There are two different ways to pass the specification-architecture pair to ReaSyn. The architecture definition file, as described in section 8.1, is preceeded by the `-a` option. The specification can be passed either as an LTL formula (`-f` option, c.f section 8.2) or as a deterministic Büchi automaton (`-b` option, c.f section 8.2.2).

`# ./ReaSyn [options] -a` *<architecture file>* `-f` *<formula file>*

`# ./ReaSyn [options] -a` *<architecture file>* `-b` *<automaton file>*

If ReaSyn is run without an architecture definition or a specification it will ask the user to provide the missing parts during runtime. There are several options, which could be used with ReaSyn:

- `--optimize1` enables the dead end optimization for distributed games [Mau05].
- `--optimize2` enables the winning region optimization for distributed games [Mau05].
- `--no-code` disables the code generation.
- `--no-buchi-reduction` disables the reduction of the used Büchi automata.
- `--gdl` enables the graph output for the used games. The descriptions are written into `.gdl` files which could be displayed using AiSee [aAIG].
- `--no-parity-approximation` enables the correct transformations of the "all-path parity" winning condition into a parity winning condition.
- `--lazy-approximation` enables the approximation of the "all-path parity" condition expecting the worst case for the Environment. By default the best case for the Environment is taken as approximation.
- `--no-winning-region-optimization` disables the reduction of the winning region using simulation.

ARCHITECTURE:

```
// -x-> |P| -y-> |Q| -z->

Process P;
Process Q;

Signal x 0 1;
Signal y 0 1;
Signal z 0 1;

Input P x;
Input Q y;

Output P y;
Output Q z;
```

FORMULA:

```
[]((x && y)->z)
```

FIGURE 8.8 A Simple Problem

- `--random-strategy` Disables the mimization of the generated finite state program. Instead a strategy is picked randomly to generate the finite state program. It is faster than finding a minimal strategy but may yield bigger programs.

8.4 Code Output

The primary goal of ReaSyn is to generate a minimized program, which satisfies a given specification. Formally, the output is a deterministic finite state transition system for each process of the architecture. It is represented as PROMELA program.

An example of a simple problem is given in Figure 8.8. As can be seen, the architecture is a three-process pipeline with the requirement (LTL specification) that: ”It is always the case that whenever `x` and `y` equal 1 at the same time, then `a` does too”. Figure 8.9 contains the program generated by ReaSyn. The signals are implemented as channels with capacity one. Channels with capacity one can hold at most one value. The value has to be read from the channel, before a new value can be assigned. Generally, the capacity of a channel equals the number of its readers.

GENERATED PROGRAM:

```
chan xChan = [1] of { int };
chan yChan = [1] of { int };
chan zChan = [1] of { int };

int x = 0;
int y = 0;
int z = 0;
bit check = 0;

active proctype P() {
byte state = 0;
bit xRead = 0;
do
:: (state == 0) ->
        xRead = 0;
        do
        :: xChan?x -> xRead = 1;
        :: (xRead) -> break;
        od;
        if
        :: ((x == 0)) ->
                yChan!1;
                state = 0;
        :: ((x == 1)) ->
                yChan!0;
                state = 0;
        fi;
od
}

active proctype Q() {
byte state = 0;
bit yRead = 0;
do
:: (state == 0) ->
        yRead = 0;
        do
        :: yChan?y -> yRead = 1;
        :: (yRead) -> break;
        od;
        if
        :: ((y == 0)) ->
                zChan!1;
                state = 0;
        :: ((y == 1)) ->
                zChan!0;
                state = 0;
        fi;
od
}

active proctype env() {
do
:: xChan!1; zChan?z; check = 1; check = 0;
:: xChan!0; zChan?z; check = 1; check = 0;
od
}
```

FIGURE 8.9 Output of the simple problem

PROMELA was chosen as output language because it is well known and widely spread. During the validation phase of ReaSyn the SPIN (**s**imple **P**RO-MELA **in**terpreter) model checker was used to prove that the generated programs where correct. SPIN can be used to prove that a PROMELA program satisfies arbitrary LTL formulas.

However, some limitations are acknowledged. SPIN is a model checker for asynchronous PROMELA programs. ReaSyn, in contrast, produces a finite state PROMELA program, which is meant to be a synchronous model. It assumes that all processes compute their output in one atomic step. In order to adapt the formula to the asynchronous setting, a formula translator was created during the validation phase of ReaSyn. The generated program incorporates a flag `check`, which is `true`, if and only if all processes have finished their computations and are awaiting new input. The following rules define how to adapt the property to the synchronous setting of the generated program.

Original property	Adapted property
$\Box\varphi$	$\Box(\mathsf{check} \rightarrow \varphi)$
$\Diamond\varphi$	$\Diamond(\mathsf{check} \rightarrow \varphi)$
$\varphi_1 \mathsf{U} \varphi_2$	$(\mathsf{check} \rightarrow \varphi_1)\, \mathsf{U} (\mathsf{check} \rightarrow \varphi_2)$
$\varphi_1 \rightarrow \varphi_2$	$(\mathsf{check} \wedge \varphi_1) \rightarrow \varphi_2$
$\varphi_1 \vee \varphi_2$	$\mathsf{check} \rightarrow (\varphi_1 \wedge \varphi_2)$
$\varphi_1 \wedge \varphi_2$	$\mathsf{check} \rightarrow (\varphi_1 \wedge \varphi_2)$
$\neg\varphi$	$\mathsf{check} \rightarrow \neg\varphi$

Note that any LTL property containing the X (next) operator cannot be easily proven using the SPIN model checker. SPIN provides no support for the X operator. In order to prove properties containing a X operator, the program has to be adapted with toggle variables for every next operator. A validation of the generated program is not nescessary. Its correctness follows from its construction. Therefore, the complicated construction needed to incorporate the next operator is omitted.

Chapter 9

Internal Interfaces

As aforementioned, ReaSyn is composed of four parts. The first part is concerned with translating formulas and architectures into games. Before architectures are encoded as games, they are transformed, if possible, into pipeline architectures. These transformations are necessary, as the algorithm presented in [MW03] was designed only for this class of architectures. In the second part of ReaSyn, as described in [Mau05], the games for formula and architecture are combined into one distributed game. This part therefore includes the algorithms used to simplify and to solve these distributed games. It is supported by the third part, which provides functionality to convert the "all-path parity" condition into an ordinary parity winning condition. This part is also covered by the present work. The parts two and three are closely related. However, part three uses the automata transformations introduced to transform the specification. Therefore, the transformation of winning conditions was spilted off from part two. The winning strategies and winning regions produced by the second part are then passed to the fourth part of ReaSyn. There, the strategies are used to generate PROMELA code, which fulfills the specification-architecture pair. The design of ReaSyn follows the layer design pattern (see Figure 9.1). The first layer covers the parts one and three. The remaining parts two and four each represent a layer. Layer one first reads the architecture and the specification provided by the user. After transforming them into two-player games, it passes the games to the second layer. Additionally, it provides functionality to the second layer to transform a game with the "all-path parity" winning condition into an ordinary parity game. In the second layer, the games are combined into a distributed game [MW03]. If the given problem has a solution, layer two will create a winning strategy for Player, which is passed on to the next layer. Finally, the third layer creates a PROMELA program implementing the architecture and satisfying the specification.

As this thesis covers layers one and three, there are two different parties the layers communicate with. The user interface was already described in chapter 8. The interface to the second layer is defined in the remainder of this chapter.

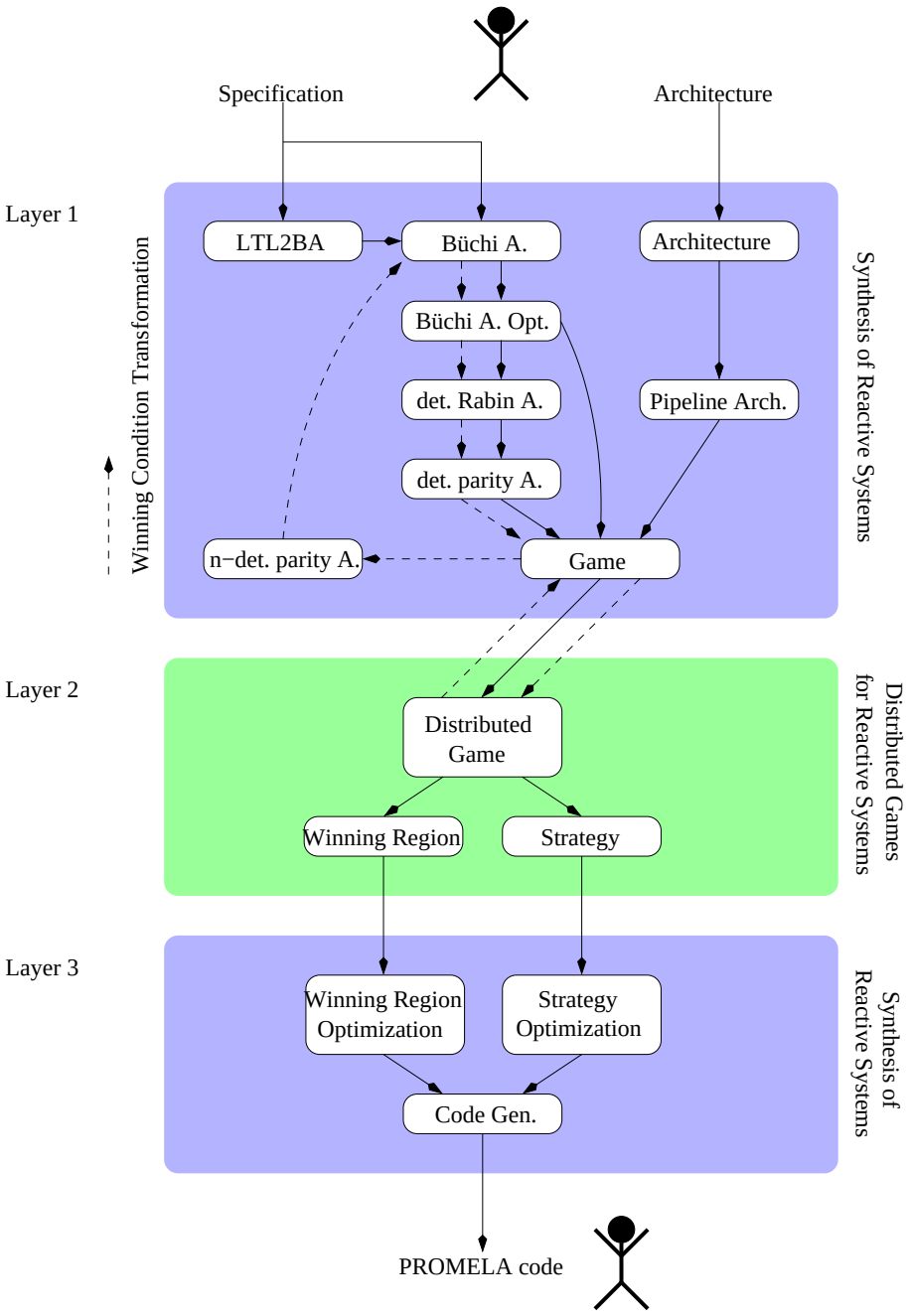


FIGURE 9.1 Structure of ReaSyn

It defines the properties of the data exchanged between the three layers.

- **Architecture Games**: The architecture games are bipartite two-player games of Player versus Environment. The architecture has to be a strictly hierarchical architecture without an information fork. Each of the games simulates a single process p of the architecture. The Player chooses the function $f_p : D(I(p)) \rightarrow D(O(p))$, while the Environment chooses the input value $x \in D(I(p))$ of the process. The games only simulate the architecture and therefore have no winning conditions.

- **Specification Game**: The specification game is a two-player parity game of Player versus Environment simulating a deterministic parity automaton, which accepts all sequences that fulfill the specification. In the game, the Environment chooses the valid atomic propositions while the player chooses the transition the automaton would take. Since the automaton is a deterministic one, the Player moves are determined.

- **Winning Region**: The winning region contains all nodes of the reduced distributed game from which every play is winning for Player. The reduced distributed game is a two-player game.

- **Strategie**: The strategy is a non-deterministic memoryless Strategy $s : \mathrm{node} \rightarrow 2^{\mathrm{node}}$, which maps every Player node of the Players' winning region of the reduced distributed game to a set of successors. Every play $\pi = n_0, n_1, n_2, \ldots$ with $n_i \in s(n_{i-1})$ and the initial node n_0 of the distributed game is winning for Player.

- **Winning Condition Transformation**: The winning condition transformation expects as intput an initialized "all-path parity" game and the initialized parity game, which is referred to by the "all-path parity" condition. It returns a two-player parity game which has a winning strategy for Player from the initial node, if and only if Player has a winning strategy from the initial node of the "all-path parity" game.

Chapter 10

Implementation

10.1 Specification Transformations

In ReaSyn the specification is transformed into a game. The purpose of the specification game is to ensure that the system behavior generated by the architecture games satisfies the specification. If the user has provided a deterministic Büchi automaton, the automaton is directly transformed into a game satisfying the requirements of the interface to [Mau05] using the transformation from section 6.2. In case the user has provided an LTL formula, the specification is first transformed into a very weak alternating automaton, then into a generalized Büchi automaton, and finally into a Büchi automaton. These transformations are done by the tool LTL2BA [GO01]. If the resulting Büchi automaton is deterministic, the above mentioned tranformation into a game follows. Otherwise, the non-deterministic Büchi automaton is first transformed into a deterministic Rabin automaton, which is subsequently converted into a deterministic parity automaton. The parity automaton is then encoded into a two-player parity game using the transformation from Section 6.2.

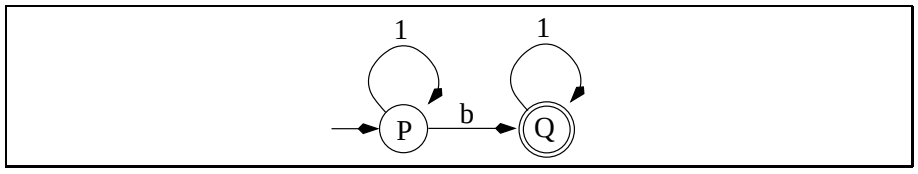


FIGURE 10.1 Büchi automaton for the formula $\diamond b$ generated by LTL2BA[GO01]

10.1.1 Fast LTL to Büchi Automata

The first three transformations, namely LTL to very weak alternating automaton to generalized Büchi automaton to Büchi automaton, are performed by the tool LTL2BA [GO01]. It provides an efficient implementation that produces optimized Büchi automata. In the present work, LTL2BA has been slightly modified to accept arithmetic expressions as atomic propositions and to create the Büchi automaton as an instance of the ReaSyn datastructures. Figure 10.1 shows the non-deterministic Büchi automaton for the formula $\Diamond b$, generated by LTL2BA. Gastin and Oddoux [GO01] optimize the automata by removing transitions, which are implied by other transitions and by merging equivalent states of the automata.

- A transition $t_1 = (q, \sigma_1, q_1)$ implies a transition $t_2 = (q, \sigma_2, q_2)$, if

In a VWAA	$\sigma_2 \subseteq \sigma_1 \wedge q_1 \subseteq q_2$
In a GBA	$\sigma_2 \subseteq \sigma_1 \wedge q_1 = q_2 \wedge \forall T \in ACC : t_2 \in T \Rightarrow t_1 \in T$
In a BA	$\sigma_2 \subseteq \sigma_1 \wedge q_1 = q_2$

- Two states q_1 and q_2 are equivalent, if

In a VWAA	$\delta(q_1) = \delta(q_2) \wedge q_1 \in ACC \Leftrightarrow q_2 \in ACC$
In a GBA	$\delta(q_1) = \delta(q_2) \wedge \forall (\sigma, q') \in \delta(q_1), T \in ACC :$ $(q_1, \sigma, q') \in T \Leftrightarrow (q_2, \sigma, q') \in T$
In a BA	$\delta(q_1) = \delta(q_2) \wedge q_1 \in ACC \Leftrightarrow q_2 \in ACC$

Although these reductions are performed by LTL2BA, ReaSyn additionally reduces the Büchi automaton using delayed simulation [EWS01]. State space reduction through delayed simulation is more efficient in decreasing the state space of a Büchi automaton than the reductions used in LTL2BA. Experimental results have shown that even a single node can have an enormous influence on the size of the transformed automata.

The implementation of the remaining transformations, that is the determinization of the Büchi automaton and the conversion of the resulting deterministic Rabin automaton into a deterministic parity automaton, follow the same idea. Begining with the starting nodes, all possible successors are computed. The successors are added to the automaton and then the successors of the newly added nodes are determined etc. This algorithm computes the reachable subgraph rooted in the initial nodes.

10.1.2 Game Encoding

The encoding of an automaton into a parity game follows from the construction from Section 6.2. First, the Environment nodes are created. For each transition

$\mathcal{P} = (V, \Sigma, \delta, v_0, \alpha)$
$\mathcal{G} = ((V_{player}, V_{env}, E), \chi)$

for $v \in V$ **do**
    V_{env}->add(v)
    $\chi(v) := \alpha(v)$
done

for (v, σ, w) with $\delta(v, \sigma) = w$ **do**
    V_{player}->add((v, σ))
    $\chi((v, \sigma)) := \alpha(v)$
    E->setTransition($(v, (v, \sigma))$)
    E->setTransition($((v, \sigma), w)$)
done

initialize the game with the node $v_0 \in V_{env}$

FIGURE 10.2 Specification to Game Encoding Algorithm. $\mathcal{P}$ is a parity automaton, $\mathcal{G}$ the created parity game.

$t = (q, \sigma, q')$ from a node q to q' labeled with σ, a new Player node $e = (q, \sigma)$ is created. Furthermore, transitions from q to e and from e to q' are added. The parity condition is updated every time a new node is added to the game. Figure 10.2 shows the algorithm used for the transformation of a parity automaton. The algorithm for Büchi automata works alike except for the definition of χ. For a deterministic Büchi automaton $\mathcal{B} = (V, \Sigma, \delta, v_0, F)$, a node v has parity 0, if $v \in F$ otherwise it has parity 1. This definition extends to player nodes as in the algorithm introduced in Figure 10.2.

10.2 Architecture Transformations

In order to obtain the architecture games, the architecture has to be transformed into a strictly hierarchical acyclic architecture without an information fork. The algorithms used in [Mau05] can be applied to this class of architectures only. Finkbeiner and Schewe [FS05] introduced transformations, which result in such an architecture. The transformations consist of removing idle processes, clustering equally informed processes, and removing feedback edges. They were introduced in detail in Section 5.1. Additionally, a transformation that encodes strictly hierarchical architectures into pipeline architectures was introduced.

Once transformed, the architecture processes can be encoded as games. First, the domains of the input signals are combined into a single domain by building the cartesian product of the input domains. The output domains are combined likewise. Every element of the combined input domain is added as a new

Player node to the process game. Second, all total functions from the combined input domain to the combined output domain are added as new Environment nodes. Finally, all possible transitions between Player and Environment nodes are added. The resulting game is bipartite and simulates all possible behaviors of the process. The Player chooses the behavior of the process, while the Environment determines the input of the process. This encoding meets the assumptions of the interface (c.f. Chapter 9).

10.3 Code Generation

Code generation is covered in the third part of ReaSyn (c.f. Section 2.3). As mentionend before, Maurer [Mau05] creates a distributed game [MW03] from the architecture and specification games and solves it. Provided that the Player wins the distributed game, he passes on a non-deterministic winning strategy and the winning region for the Player. Formally, the non-deterministic strategy is represented as a function $F : V \rightarrow 2^V$ that maps a node to a set of possible successor nodes. Winning a distributed game stands for having a winning strategy for the initial node of the distributed game. Using this non-deterministic strategy, one can construct a subgraph of winning region for Player. It is constructed using the transitions for Environment nodes as in the distributed game, but reducing the possible successors of a Player node v to $F(v)$, where F is the non-deterministic winning strategy. The problem of generating code from this winning region and the non-deterministic strategy is divided into two parts. First, one has to determine the strategy. As before, the strategy identifies a subgraph of the winning region containing the initial node of the distributed game, in which the Player moves are determined. The second part of the code generation is to create PROMELA code from a given strategy. In fact, the code is generated using both the Player and the Environment nodes. The latter represent the functions computed by individual processes of the architecture and are therefore equal to the number of states in the finite state program. The Player nodes serve to designate the successor states of the generated finite state programs (one for each process of the architecture) depending on the input of the process.

In order to minimize the number of states in the generated program one can use two starting points for optimizations. First, one tries to reduce the winning region by merging equivalent nodes. Second, the determinization of the non-deterministic strategy offers possibilities to minimize the number of states in the generated program.

10.3.1 Optimization of the Winning Region

ReaSyn is sought to generate an optimized program for each process of the architecture. As mentioned before, the number of Environment processes occuring in the subgame represented by a strategy, is equal to the number of states in the finite state programs. The size of the resulting programs can therefore be reduced by minimzing the number of nodes in the winning region. In the distributed game, nodes of the Environment player represent the functions computed by the architecture processes. The Player nodes represent the input values chosen by the Environment. The winning region can be reduced by merging equivalent nodes. Obviously, Player nodes cannot be merged with Environment nodes and vice versa. The resulting winning region is the quotient winning region with respect to the equivalence relation

$$p \sim q \Leftrightarrow \begin{cases} p \sim_p q & , p, q \text{ are Player nodes} \\ p \sim_e q & , p, q \text{ are Environment nodes} \end{cases},$$

which is composed of an equivalence relation $\sim_p$ for Player nodes and $\sim_e$ for Environment nodes, respectively.

The equivalence realtion $\sim_p$ for Player nodes is defined using ordinary simulation [EWS01]. In general, ordinary simulation does not maintain the winner of a game, but since every path in the winning region is winning for the Player, it is sufficient for the node space reduction of the winning region.

In order to determine whether a node q can simulate a node q', a parity game with players Spoiler and Duplicator is played. At round zero, two pebbles (*red*, *blue*) are placed on q and q', respectively. The Spoiler moves the *red* pebble, while the Duplicator moves the *blue* pebble. At round i, let *red* be on q_i and *blue* on node q'_i. The moves of the players are as follows:

1. Spoiler chooses a triple (q_i, e_i, q_{i+1}) with $e_i \in F(q_i) \wedge q_{i+1} \in F(e_i)$ and puts *red* on q_{i+1}.
2. The Duplicator has to respond by choosing a triple (q'_i, e'_i, q'_{i+1}) with $(e'_i \in F(q'_i) \wedge q'_{i+q} \in F(e'_i))$. Additionally, e_i and e'_i have to represent the same functions. If no such triple exists the game halts and Spoiler wins.

Formally, the game $\mathcal{G}^o = ((V_{Duplicator}, V_{Spoiler}, E), \chi)$ is constructed from the winning region W and the set of winning strategies F as described in [EWS01]:

- $V_{Duplicator} = \{(q, q', e) \mid q, q' \text{ are Player nodes and } e \text{ is an Environment node} \wedge q \in F(e)\}$

- $V_{Spoiler} = \{(q, q') \mid q, q' \in W$ and both are Player nodes $\}$
- $E = \{((q_1, q_1', e), (q_1, q_2')) \mid \exists e'.e, e'$ represent the same functions
  $\wedge\, e' \in F(q_1') \wedge q_2' \in F(e')\}$
  $\cup \quad \{((q_1, q_1'), (q_2, q_1', e)) \mid e \in F(q_1) \wedge q_2 \in F(e)\}$
- $\chi(v) = 0, v \in (V_{Duplicator} \cup V_{Spoiler})$

This game is easily solved. For Spoiler, the only way to win the game is to move into a dead end for Duplicator. The algorithm used for finding the winning regions and strategies for both players is described in detail in [Mau05]. The above mentioned construction leads to a game, in which the Duplicator has a winning strategy from a node (q, q'), if and only if q' can simulate q. Given the winning region W for Duplicator, the equivalence relation $\sim_p$ is defined as follows:

$$q \sim_p q' \Leftrightarrow ((q, q') \in W \wedge (q', q) \in W)$$

The equivalence relation $\sim_e$ for Environment nodes is much simpler. Two Environment nodes e_1 and e_2 are equal, if and only if

1. e_1 and e_2 represent the same functions,
2. $F(e_1) = F(e_2)$.

The reduction itself is obtained by building the quotient winning region of the distributed game with respect to $\sim$. Figure 10.3 shows the algorithm used by ReaSyn to optimize the winning region of the Player in the distributed game. The Player nodes are merged first, because it yields a better reduction for Environment nodes.

10.3.2 Strategy Determinization

A strategy for Player is a function $f_{Player} : V_{Player} \rightarrow V$ that maps a Player node to a successor. It defines the behavior of the underlying distributed system. Different strategies lead to different implementations. ReaSyn is interested in finding a small program in terms of number of states of the finite state program. A first step to reduce the size of the generated program was presented in the previous section. This section covers the algorithm used by ReaSyn to find a strategy for Player that involves the smallest possible number of Environment nodes. The problem of choosing a strategy f_{Player} from a set of strategies $F : V \rightarrow 2^V$ is to pick a designated successor for every Player node v from $F(v)$. A strategy for Player identifies a subgraph of the winning region for Player in the distributed game in which the Players' moves are determined.

Let W_{Player} and F be the winning region for Player and the set of strategies, respectively. $[v]_\sim$ denotes the equivalence class of v with respect to $\sim$. V_σ are the σ-nodes from the distributed game ($\sigma \in \{Environment, Player\}$. $W[x := y]$ denotes a substitution of x by y in W.)

```
G^o := constructGame(W, F)
W_Duplicator := solve(G^o)

~_p := obtainPlayerRelation(W_Duplicator)
for v ∈ W_Player ∩ V_Player do
    W_Player[v := [v]_~p]
    F'(v) := ⋃_{v'∈[v]_~p} F(v')
done

~_e := obtainEnvRelation(W_Player)
for v ∈ W_Player ∩ V_Environment do
    F'(v) := ⋃_{v'∈[v]_~e} F(v')
done
```

FIGURE 10.3 Algorithm used by ReaSyn to optimize the size of the winning region

F is a strategy, `init` is the initial node of the distributed game.

```
maxNumber = 1
S = ∅

while true do
   // pick a new SubGraph with maxNumber
   // Envnodes initialized with init.
   S = nextSubGraph(maxNumber, init)
   if hasNoDeadEnds(S, F)
      then break
   if noMoreSubGraphs(maxNumber, init)
      then maxNumber++
done

f := obtainStrategy(S)
```

FIGURE 10.4 Breadth-first search algorithm used by ReaSyn to find a small subgame and the related strategy.

ReaSyn realizes the strategy determination with a breadth-first search algorithm. Initially, it starts with an empty subgame. Every play has to begin at the initial node of the distributed game. Hence, the initial node is added to the subgame. It then enumerates all subgames with one Environment node. One by one, every successor of the initial node is added to the subgame, as well as all its successors. Subsequently, the subgame is checked for dead ends. If a dead end is found, the Environment node and its successors are removed from the subgame and the next Environment node is tested as described above. If no subgame without dead ends exists, the number of possible Environment nodes in the subgame is increased and the procedure is repeated. Figure 10.4 shows the algorithm used in ReaSyn to determine a strategy. The function `nextSubGame(maxNumber, init)` enumerates all subgames with `maxNumber` Environment nodes that are initialized with `init`. `hasNoDeadEnds(`$\mathcal{S}$`,`F`)` returns true, if $\mathcal{S}$ contains no dead ends with respect to F, while `noMoreSubGames(maxNumber, init)` returns true, if `nextSubGame` has enumerated all subgames with size `maxNumber` that are rooted in `init`. If a subgame is found, it corresponds to a strategy f_{Player}. This strategy is used to generate the finite state PROMELA program.

The algorithm presented above will produce a strategy with the smallest number of Environment nodes possible. This may take a lot of time. Therefore, ReaSyn provides a commandline option to skip the breath-first search for a best strategy. Instead, beginning with the initial node of the distributed game a successor is randomly chosen and added to the subgraph until no dead ends are left. This procedure is much faster but will produce bigger programs than the breath-first search.

10.3.3 Generating Code

Before describing the code generation in detail, some preliminary properties of distributed game nodes need to be clarified.

Distributed Game Nodes

The distributed game is constructed from the architecture and specification games, so-called local games. The distributed game assumes that the Environment has complete information. Hence, a distributed game with n local games is a game of $n + 1$ players. The distributed game plays all local games, simultaneously. For a detailed construction refer to [MW03, Mau05]. Due to the "divide" and "glue" operations, the distributed game of $n + 1$ players is transformed into a two-player game. The "divide" operation merges two local games, while the "glue" operation combines nodes of the first local game in order to establish determinism for Environment positions. During the "divide" operation nodes of the first and the n-th local game are merged into a

`PairNode`, which is a pair of two nodes; the first component of which represents a node from the first local game and the second a node from the n-th local game. The "glue" operation combines successors of Environment nodes into a `SetNode`. As indicated by its name, a `SetNode` represents a set of nodes of the first local game. Every member of the set is a pair. The second component of the pair states the current node, while the first component is a reference to the predecessor of the node. In order to find a representative of every `SetNode` in a "glued" game, one picks a pair from every set in a way that for every two consecutive pairs p_1, p_2 in a play of the game the first component of p_2 matches the second component of p_1. As "divide" and "glue" are excecuted several times, `SetNode`s and `PairNode`s can be arbitrarily composed. But in a distributed game of n local games, there are $n-1$ `PairNode`s and at most $n-2$ `SetNode`s nested.

Code Generation

After a minimized strategy is found, the code can be generated. As mentionend before, Environment nodes represent the functions chosen by the player, while Player nodes represent the different input values chosen by the Environment.

The number of states of the program is given by the number of Environment nodes in the subgraph represented by the strategy. Therefore, the Environment nodes are numbered. The initial state of the generated finite state program is represented by the successor of the initial node of the distributed game. Recall, that in the subgraph identified by the chosen strategy, all Player moves are determined. In the following, an implementation is generated for every process from the original architecture. For a state, which is an Environment node v, the functions g_i are extracted. The functions are the Environment nodes from the local architecture games. They are represented as nested `PairNode`s and `SetNode`s. In order to create an implementation for a process, a component matching the process's signature has to be identified. Due to the architecture transformation, none of the functions need to exactly match the incoming or the outgoing signals of the process. In order to find a matching function, the output domain of the process has to be part of the output domain of the function. In other words, the function computes at least the values for the signals of the process output domain. Signals that occur in the input domain of the process but not in the input domain of the function are feedback signals. The function is used to create the implementation for the process. In the generated program, the process first reads the information of all its input signals except for the feedback signals. It then creates the output for its output signals according to the before determined function. Depending on the input value, a successor state in the finite state program is determined. Environment nodes have a successor for every input value. The successor state in the finite state program for an input value i corresponds to the successor of the Player node corresponding to the input value i. After processing the input, the feedback

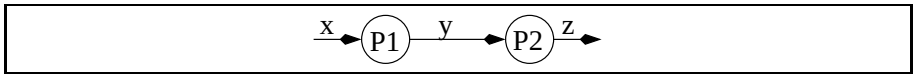


FIGURE 10.5 Four process two-way ring architecture before and after transformation

signals are read. This procedure is repeated until every process has an implementation for every Environment node of the subgame. Finally, the implementation for the environment process is generated. The environment process non-deterministically chooses values for the signals under its control.

Figure 10.5 shows a two process pipeline. Together with the specification $(x \rightarrow \Box(\neg z)) \wedge (\neg x \rightarrow \Box z)$ the Distributed Synthesis Problem is solved. After optimization of the winning region the subgraph in Figure 10.7 is obtained. The subgraph contains three Environment nodes, therefore the generated program has three states. It is shown in Figure 10.6. The function used to create the implementation of the process `P1` is `f4`. As can be seen in the example, `P1` only implements one function in all three states of the program. It indicates that optimizations on the process level would further reduce the generated program. However, this optimization is not covered by the present work. In the example displayed in Figure 10.7 there are only two possible threads for the first component. The function `f4` is used to implement `P1`'s behavior in all of the three states. The second component is used to implement the program for `P2`. In order to determine the successor state in the finite state program the Player nodes are used. Each player node represents an input value. The successor state for an Environment node v and an input value i is the successor of a successor of v representing i.

In the generated PROMELA program the signals are represented by channels (**`chan`**). They have a limited capacity indicating how many values the channel can hold at a time. Every time a value is written to the channel (**`!`** operator), the number of values on the channel increases. Reading from the channel (**`?`** operator) decreases the number of values on the channel. If the channel is empty, the reading process will pause its execution until a value is written to the channel. In ReaSyn, multiple processes can read a signal, therefore the capacity of the corresponding channel is set to the number of readers n and every value written to the channel is copied n times. The channel declarations are followed by variable declarations. The variable `check` is true whenever a computation cylcle of the distributed system is completed (cf. Chapter 8). Additionally, variables for every signal are declared to ease the access to the channel values in the SPIN model checker. Subsequently, the processes are defined (**proctype**). For every state of the finite state program all input channels – except for the feedback channels – are read. Subsequently, the output values are computed according to the matching function. Finally, the feedback channels are read. For a complete introduction to PROMELA refer to [Hol03].

```
chan xChan = [1] of {int};
chan yChan = [1] of {int};
chan zChan = [1] of {int};

int x = 0;
int y = 0;
int z = 0;
bit check = 0;

active proctype P1() {
byte state = 0;
bit xRead = 0;
do
:: (state == 0) ->
        xRead = 0;
        do
        :: xChan?x -> xRead = 1;
        :: (xRead) -> break;
        od;
        if
        :: ((x == 0)) ->
                yChan!1;
                state = 2;
        :: ((x == 1)) ->
                yChan!0;
                state = 1;
        fi;
:: (state == 1) ->
        xRead = 0;
        do
        :: xChan?x -> xRead = 1;
        :: (xRead) -> break;
        od;
        if
        :: ((x == 0)) ->
                yChan!1;
                state = 1;
        :: ((x == 1)) ->
                yChan!0;
                state = 1;
        fi;
:: (state == 2) ->
        xRead = 0;
        do
        :: xChan?x -> xRead = 1;
        :: (xRead) -> break;
        od;
        if
        :: ((x == 0)) ->
                yChan!1;
                state = 2;
        :: ((x == 1)) ->
                yChan!0;
                state = 2;
        fi;
od
}

active proctype P2() {
byte state = 0;
bit yRead = 0;
do
:: (state == 0) ->
        yRead = 0;
        do
        :: yChan?y -> yRead = 1;
        :: (yRead) -> break;
        od;
        if
        :: ((y == 0)) ->
                zChan!0;
                state = 1;
        :: ((y == 1)) ->
                zChan!1;
                state = 2;
        fi;
:: (state == 1) ->
        yRead = 0;
        do
        :: yChan?y -> yRead = 1;
        :: (yRead) -> break;
        od;
        if
        :: ((y == 0)) ->
                zChan!0;
                state = 1;
        :: ((y == 1)) ->
                zChan!0;
                state = 1;
        fi;
:: (state == 2) ->
        yRead = 0;
        do
        :: yChan?y -> yRead = 1;
        :: (yRead) -> break;
        od;
        if
        :: ((y == 0)) ->
                zChan!1;
                state = 2;
        :: ((y == 1)) ->
                zChan!1;
                state = 2;
        fi;
od
}

active proctype env() {
do
:: xChan!1; zChan?z; check = 1;
    check = 0;
:: xChan!0; zChan?z; check = 1;
    check = 0;
od
}
```

FIGURE 10.6 Code output for the Distributed Synthesis Problem with the architecture from Figure 10.5 and the specification:
$(x \rightarrow \Box(\neg z)) \wedge (\neg x \rightarrow \Box z)$

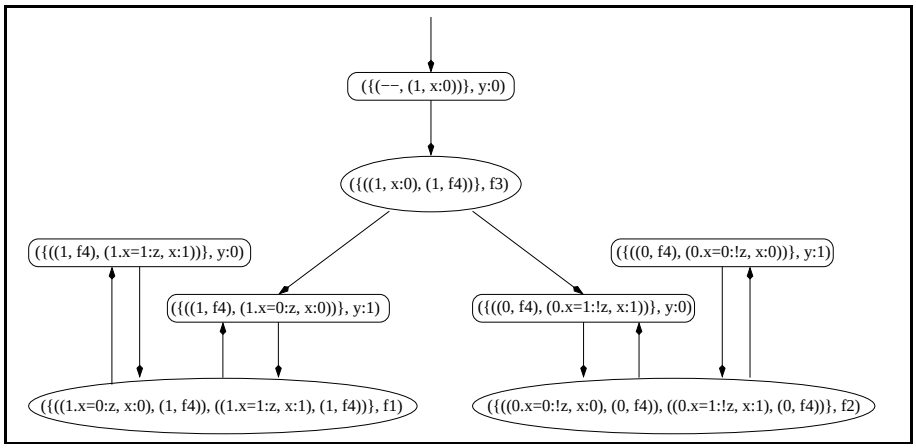


FIGURE 10.7 The optimized subgraph for the architecture from Figure 10.5 and the specification $(x \rightarrow \Box(\neg z)) \wedge (\neg x \rightarrow \Box z)$. Player nodes are rectangular, Environment nodes are elliptic.

Chapter 11

ReaSyn in Numbers

As ReaSyn is a first approach to solve the Distributed Synthesis Problem, there are no programs to compare ReaSyn to. Nevertheless, there are several subproblems, which have been realized by other programs. The program LTL2BA [GO01] is used by ReaSyn to transform the specification into a non-deterministic Büchi automaton. In order to create a game, a deterministic automaton is needed. ReaSyn uses Safra's [Saf88] construction for the determinization of the Büchi automaton. It implements the algorithm from [THB95], which is also used in HSIS [A A94].

11.1 Optimization of Büchi Automata

The specification of the Dining Philosophers Problem is a ilustrative example for the implications of the state space reduction for Büchi automata. In the Dining Philosphers Problem a number of philosophers sit on a round table. Only one chopstick is placed between two philosophers. A philosopher

Implications of Büchi Automata Reduction

	BA Size	opt. BA Size	Time	opt. Time	Game Size	opt.Game Size
DP_2	5	4	<1s	<1s	1107	99
DP_3	6	5	0:0:09	0:0:01	24288	1683
DP_4	7	6	0:01:10	0:0:06	125388	8385
DP_5	8	7	0:54:07	0:0:41	645867	43092
DP_6	9	8	6:31:51	2:48	1576450	103525

TABLE 11.1 Times are measured in hours:min:sec. Game size is the size of the specifi cation game. The opt. colums show results with enabled Büchi automata reduction

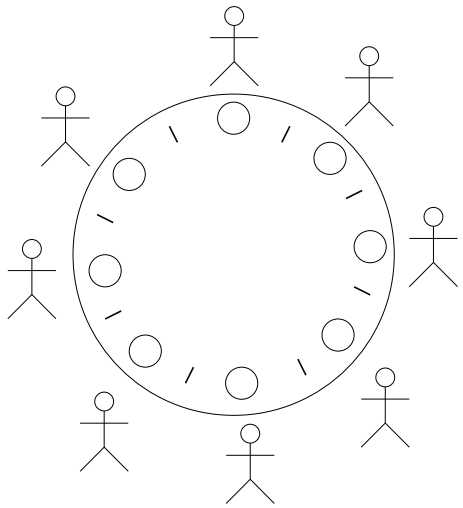


FIGURE 11.1 Eight Dining Philosophers

controls the chopstick on his right side. He can eat if he can access two chopsticks. This can happen, if the philosopher on his left side decides not to eat. The Dining Philosophers Problem is to find a behaviour for the philosophers, which allows every philosopher to eat eventually. Figure 11.1 shows the situation of the Dining Philosophers Problem with eight philosophers. For ReaSyn one of the philosophers is controlled by the Environment. The specification of the Dining Philosophers Problem there are two boolean variables for each philosopher. The first determines whether the philosopher is eating, the second determines whether the philosopher uses the chopstick under his control. The specification then states: *If the Environment philosopher infinitly often releases his chopstick, all philosophers will eat infinitly often.* The LTL formula for n philosophers looks like:

$$\Box\Diamond(\neg ch_{env}) \rightarrow \bigwedge_{1\leq i\leq n} (\Box(eat_{phil_i} \rightarrow (\neg ch_{phil_{i-1}} \wedge ch_{phi_i})) \wedge \Box\Diamond(eat_{phil_i}))$$

eat_{phil} states that $phil$ is eating while ch_{phil} is true, if $phil$ uses his chopstick. The Environment philosopher is $phil_0$. Table 11.1 shows the results for the transformation of the specification with and without the state space reduction of Büchi automata using delayed simulation. DP$_i$ is the specification for i philosophers. Although delayed simulation reduces the Büchi automata by only one node, the implications for the following transformations are enormous. The game mentionend in Table 11.1 is the specification game. The size of the game is its number of nodes.

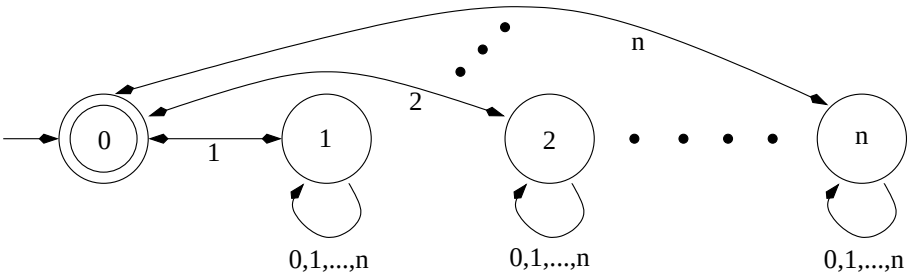


FIGURE 11.2 Transition Graph of the Büchi automaton $\mathcal{M}_n$

Determinization Efficiency

Büchi Automaton	ReaSyn		OmegaDet	
	states	pairs	states	pairs
$\mathcal{M}_1$	4	1	7	1
$\mathcal{M}_2$	20	2	33	2
$\mathcal{M}_3$	283	5	385	5
$\mathcal{M}_4$	13598	6	13601	7
$\mathcal{M}_5$		†	1059057	9

TABLE 11.2 The Table compares the number of states and Rabin pairs produced by ReaSyn and OmegaDet. †: memory exceeded

11.2 Determinization of Büchi Automata

Another approach to automata determinization is the program OmegaDet [ATW05]. The table below compares the performance of the determinization procedure of ReaSyn and OmegaDet. $\mathcal{M}_n$ is a Büchi automaton with $n+1$ states, the alphabet $\Sigma = \{0, \ldots, n\}$, and the transition graph shown in Figure 11.2. The size mentionend in the table represents the number of nodes of the deterministic Rabin automaton, while pairs stand for the number of Rabin pairs in the winning condition of the automaton. As can be seen, the resulting Rabin au-

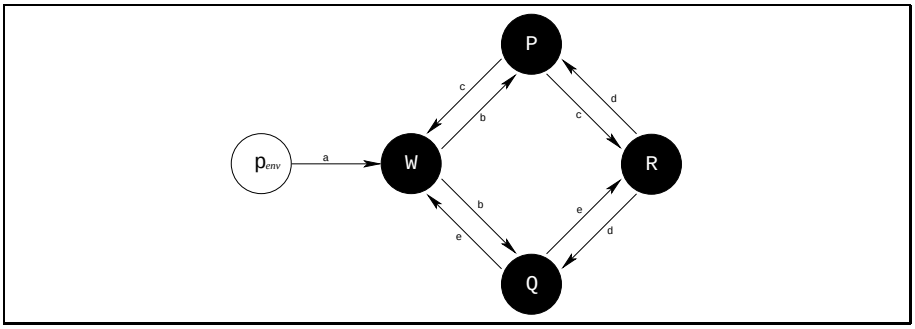


FIGURE 11.3 The architecture $\mathcal{A}_4^2$ with binary signals

Architecture Transformations

Architecture	Size	Time
$\mathcal{A}_1^2$	6	<1s
$\mathcal{A}_2^2$	12	<1s
$\mathcal{A}_3^2$	24	<1s
$\mathcal{A}_3^3$	726	<1s
$\mathcal{A}_3^4$	65800	37s
$\mathcal{A}_3^6$		†
$\mathcal{A}_4^2$	44	<1s
$\mathcal{A}_5^2$		††

TABLE 11.3 Size of the architecture games and the generation time. †: memory exceeded, ††: information fork

tomata produced by ReaSyn are much smaller than the automata generated by OmegaDet.

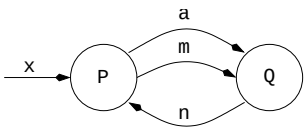
11.3 Size of Architecture Games

The size of the architecture games is directly related to the size of the distributed game used to solve the instance of the Distributed Synthesis Problem. The size of a game for process p is in $O(o^i)$ where o is the size of the output domain and i the size of the input domain. The architectures displayed in the table below are two-way ring architectures. $\mathcal{A}_m^n$ has m processes plus the Environment process and n-ary signals. Figure 11.3 shows the architecture $\mathcal{A}_4^2$. The size mentioned in the table is the summarized size of all process games.

11.4 Simple Alternating Bit Protocol

The alternating bit protocol is a connection-less protocol for uni-directional message transfer between two entities. It enhances the message with an additional bit that is flipped every time a new message is sent. The receiver answers upon reception of the message with the flipped received bit. This protocol was designed to detect corrupted messages on lossy channels. Nonetheless, a simple version of the alternating bit protocol can be synthesized by ReaSyn using the specification in Figure 11.4. Using the dead end optimization [Mau05] the problem is solved in 17 seconds. Without dead end optimization it took 45 seconds to generate the simple alternating bit protocol. During the code generation the winning region is reduced from 33 nodes to 21 nodes yielding a PROMELA program with two states. Without optimization of the winning region the program would have had three states.

ARCHITECTURE:



SPECIFICATION:

$$\Box((a \rightarrow \mathsf{X}(\neg a)) \wedge (\neg a \rightarrow (\mathsf{X}a)) \wedge \text{"}n = 1 - a\text{"} \wedge \text{"}x = m\text{"})$$

FIGURE 11.4 Alternating Bit Protocol. x contains an arbitrary message, a and m resemble the "enhanced" message send from P to Q, n contains the acknowledgement.

Part III

Discussion

Chapter 12

Evaluation

ReaSyn extends the existing theoretical works by demonstrating that distributed synthesis is possible in practice. Incorporating recent research results [MW03, FS05], ReaSyn is an insightful proof of concept, as it is the first software tool to tackle distributed synthesis. While many earlier works used an automaton-based approach to handle distributed synthesis (e.g. [KV01, MT02b]), ReaSyn applied a game theoretic approach. Primarily, the theoretical framework suggested by [MW03] was adopted and successfully altered to accommodate for LTL specifications. ReaSyn hence offers a novel starting point for both, an improved implementation and new theoretical developments.

12.1 Issues of System Size

The results presented in Chapter 11 indicate that ReaSyn performs slightly better than other implementations in determinizing automata. Nevertheless, the size of the distributed game depends on the sizes of the involved architecture and specification games. A reduction of these games therefore enables ReaSyn to solve theoretically more complex problems.

Overall, however, ReaSyn will most likely be restricted to demonstrating how abstract concepts are (un-)realizable on certain architectures, while the actual generation of complex programs is (so far) beyond ReaSyn's potential.

12.1.1 Architecture Games

Generally, the implementation of ReaSyn went beyond a mere proof of concept. While binary communication signals would have been sufficient to confirm the feasibility of the theoretical solutions to the Distributed Synthesis Problem, real-life problems are likely to require the transfer of data larger than

one bit. ReaSyn therefore attempted to offer a more sophisticated proof of concept, incorporating the option to use a finite interval of integers as signal domains, thereby investigating the effect a signal's domain size would have on ReaSyn's behavior. As one might have expected, the synthesis works reasonably well for smaller problems (for examples see Table 11.3 and [Mau05]). However, the architecture games simulate all possible functions and the number of possible functions is exponential in the size of the functions' domain sizes. Therefore, already the three-process two-way ring architecture, whose signals can hold six different values causes the synthesis to collapse (see Table 11.3). If ReaSyn is nevertheless to be used as a tool to synthesize a more complex system, the user should be aware of the exponential dependency. By cautiously abstracting the problem to be solved to the basic underlying concepts, the domain sizes of the involved signals can be significantly reduced in many cases.

12.1.2 Specification Games

Besides the size of the architecture games, the applicability of ReaSyn further depends on the size of the specification game. The more details are given in the specification, the more complex and the larger is the specification game. ReaSyn offers two possibilities to deal with this challenge. First, instead of specifying an LTL formula, the user has the option to enter a deterministic Büchi automaton. This option is obviously only reasonable, if the Büchi automaton entered by the user has fewer states than the one ReaSyn would have created. Second, ReaSyn attempts to overcome the described problem by employing a delayed simulation optimization [EWS01]. This optimization additionally significantly affects the size of the automata the Büchi automaton is transformed to. A similar optimization for the architecture games is not possible without loosing information that may be nescessary to synthesize the distributed system. Since an architecture game simulates all possible behaviors of a process, optimizing it in terms of reducing the number of transitions and/or states would also reduce the number of behaviors of the process. Thus, such an optimization could potentially eliminate the only way to synthesize a correct program.

Chapter 13

Future Directions

ReaSyn has shown that the theoretical approaches to distributed synthesis are generally feasible. Most solutions were directly adopted in the implementation of ReaSyn. While some minor changes and extensions have been introduced in the design of ReaSyn, other aspects allow for further extensions and optimizations.

13.1 Further Improvements and Optimizations

13.1.1 CTL

So far, ReaSyn accepts specifications given as LTL formula or as Büchi automaton. During the design of ReaSyn, the decision was made to use LTL rather than CTL(*) as the specification logic, because there already exists an efficient implementation [GO01] for parts of the automata transformations. A further extension to ReaSyn could be the integration of CTL* as specification language.

13.1.2 Partial Functions

Architecture games in ReaSyn model all possible behaviors of a process. This is necessary, because the processes have to anticipate the Environment's decision for an input value. Reversing the order of decision (i.e. the Environment first determines the input value of a process and only then the process decides how to react) would allow ReaSyn to reduce the size of architecture games by using partial functions instead of total ones. Here, the partial functions are only defined on one input value, which is earlier determined by the Environment. An initial test of such an extension has shown that it could constitute a

fast approximation to the solution of the Distributed Synthesis Problem.

13.1.3 Optimization for Automata

In addition to the use of partial functions, the execution time of ReaSyn could be further reduced by optimizing the Rabin- and parity automata involved in the specification transformation. In the present implementation only Büchi automata are optimized.

13.1.4 Whiteboxes

ReaSyn provides support for architectures with so called *blackbox* processes. A blackbox process is a process, whose implementation is unknown. During the implementation phase of a software tool, some components may be finished before the completion of the entire project. As the development process progresses, these components could be continuously integrated into ReaSyn's architecture description as *whitebox* processes. These are processes, whose behavior has previously been determined and is known. An extension of ReaSyn to include the possibility of whitebox processes would enable it to contribute to the software development process also at more advanced stages.

13.2 Applications

ReaSyn is the first attempt to implement a tool to synthesize distributed systems. As such, it poses a valuable contribution to the progress of theory and practice of distributed synthesis. The above described possibilities of extension illustrate, however, that much work remains to be done. Nevertheless, possible applications for a tool like ReaSyn are conceivable. For example, any software application for which the adherence to the specification is critical, such as controllers for cars, satellites, or communication protocols, could significantly benefit from a tool like ReaSyn.

List of Figures

List of Tables

Bibliography

[A A94] A Aziz, F Balarin, et al. HSIS a BDDbased environment for formal verication. In *Conference of Design Automation*, pages 454–459, 1994.

[aAIG] absInt Angewandte Informatik GmbH. aisee homepage. `http://www.aiSee.com`.

[ATW05] Christoph Schulte Althoff, Wolfgang Thomas, and Nico Wallmeier. Observations on determinization of Büchi automata. In *10th International Conference on the Implementation and Application of Automata*, 2005.

[EWS01] Kousha Etessami, Thomas Wilke, and Rebecca A. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 694–707, London, UK, 2001. Springer-Verlag.

[FS05] Bernd Finkbeiner and Sven Schewe. Unified distributed synthesis. *Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2005.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. *Computer Aided Verification*, pages 53–65, 2001.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics and Infinite Games*. Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Hol03] Gerard J. Holzmann, editor. *The Spin Model Checker*. Pearson Education. Addison-Wesley, 2003.

[KV97] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *ICTL '97*, 1997.

[KV99] Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. In *The Bulletin of Symbolic Logic*, pages 245–263, 1999.

[KV00] Orna Kupferman and Moshe Y. Vardi. μ-calculus synthesis. In *MFCS '00*, Lecture Notes in Computer Science, pages 497–507, 2000.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 389, Washington, DC, USA, 2001. IEEE Computer Society.

[Mau05] Tobias Maurer. Distributed games for reactive systems. Master's thesis, Universität des Saarlandes, maurer@react.cs.uni-sb.de, 2005.

[MS84] D. Müller and P. Schupp. Alternating automata on infinite objects: Determinacy and Rabin's theorem. In *Ecole de Printemps d'Informatique Theoretique on Automata on Infinite Words*, pages 100–107. Springer, 1984.

[MS87] D. Müller and P. Schupp. Alternating automata on infinite trees. In *Theoretical Computer Science*, pages 267–276, 1987.

[MS95] D. Müller and P. Schupp. Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. In *Theoretical Computer Science*, pages 69–107, 1995.

[MT01] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. *Lecture Notes in Computer Science*, 2076:396+, 2001.

[MT02a] P. Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 145–160, London, UK, 2002. Springer-Verlag.

[MT02b] P. Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *CONCUR '02: Proceedings of the 13th International Conference on Concurrency Theory*, pages 145–160, London, UK, 2002. Springer-Verlag.

[MW03] Swarup Mohalik and Igor Walukiewicz. Distributed games. *Conference on Foundations of Software Technology and Theoretical Computer Science '03*, pages 338–351, 2003.

[PR79] G.L. Peterson and J.H. Reif. Multi-person alternation. In *IEEE FOCS*, pages 348–363, 1979.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symp. on Foundations of Computer Science, St. Louis, Missouri*, pages 746–757, 1990.

[Rho97] S. Rhode. *Alternating Automata and the temporal logic of ordinals*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[Ros92] Roni Rosner. *Modular Synthesis Of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[Saf88] S. Safra. On the complexity of ω-automata. In *IEEE Symposisium on the Foundations of Computer Science*, pages 319–327, 1988.

[THB95] Serdar Tasiran, Ramin Hojati, and Robert K. Brayton. Language containment of non-deterministic ω-automata. In *Conference on Correct Hardware Design and Verification Methods*, pages 261–277, 1995.