

Distributed Games For Reactive Systems

Tobias Maurer
<mail@tobias-maurer.com>

Diploma Thesis
October 31, 2005

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, 31. Oktober 2005.

Acknowledgements

I would like to express my gratitude to everybody who contributed to this work by ideas and inspiration and thus helped this work to manifest.

First of all many thanks to Prof. Bernd Finkbeiner for providing the interesting topic and the helpful discussions throughout the work. Many thanks also to Sven Schewe for his comments and supervision and to Jens Regenbergh for the great collaboration in developing *ReaSyn*.

Contents

1	Introduction	2
2	ReaSyn	6
2.1	A closer look	6
2.2	Division of work	8
3	Games	9
3.1	Fundamental definitions	10
3.2	Determinacy	14
3.3	Finding a winning strategy	18
3.4	Calculating winning regions	18
3.4.1	Pseudo algorithm	18
4	Distributed Games	21
4.1	Local Game	21
4.2	Distributed Game	22
4.2.1	Examples	24
4.3	Transformations on Distributed Games	25
4.3.1	Division	25
4.3.2	Glue	27
5	Distributed Synthesis Problem	32
5.1	Architectures	34
5.1.1	Pipeline Architectures	34
5.1.2	Architecture transformations	34
5.1.3	Architecture game	35
5.2	Specification	35
5.2.1	LTL	36
5.2.2	Specification game	37
5.3	Synthesis	38
5.3.1	Encoding	38
5.3.2	Decidability	40
6	Getting to the point	42
6.1	A top down look on ReaSyn	42
6.2	Enriching the specification game	42
6.3	Constructing the Distributed Game	43

6.3.1	Building distributed game nodes	43
6.4	Simplifying the Distributed Game	45
6.4.1	Dead end removal	45
6.4.2	Removing environment winning positions	46
6.4.3	Colorspace reduction	47
6.5	Reducing the number of local players	47
6.5.1	Divide	48
6.5.2	Glue	48
6.6	Solving the reduced game	50
7	Results	54
7.1	Example 1	54
7.2	Example 2	57
7.3	Example 3	58
8	Summary	60
A	Classes	62
A.1	Class dependabilities	62
A.2	Basic datastructures	63
A.2.1	Template datastructures	64
A.3	Advanced datastructures	67
A.4	Game datastructures	68

Abstract

In system synthesis, a specification is transformed into a framework which is guaranteed to satisfy the specification. When the system is distributed, the goal is to construct the system's underlying processes.

Results on multi-player games imply that the synthesis problem for linear specifications is undecidable for general architectures. If the processes are linearly ordered and information among them flows in one direction the problem becomes decidable with non-elementary complexity. It seems plausible that the problem cannot be implemented in practice. However, LTL specifications and distributed systems are used in system design. This suggests, that the blowup does not really occur in practice. Former works concentrated on the theoretical complexity of the problem. Here a proof of concept for practical use is given by *ReaSyn*.

ReaSyn is based on the provided implementations. In case of realizability *ReaSyn* produces a valid PROMELA implementation, that simulates the distributed system. This thesis shows that for practical examples distributed synthesis is possible.

Chapter 1

Introduction

A reactive system consists of a set of processes which are connected by communication channels and receive external input signals from an environment.

Reactive systems can be defined by an architecture specifying the assembly of the processors and their signals of communication.

An interesting problem in computer science is the *Distributed Synthesis Problem*. The question is to find for a given architecture of a reactive system and a temporal specification a distributed implementation of the processes, such that their synchronous composition satisfies the specification.

The distributed synthesis problem was first considered by Pnueli and Rosner [PR90]. They proofed that it is undecidable in general. However, they also showed decidability in a setting of a hierarchical architecture named pipeline architecture with fixed communication channels between processes. Their results are based on the work by Peterson and Reif [PR79].

For the non-distributed case, the synthesis problem can be reduced to a two player game, where a player plays according to a strategy against an environment. This is called centralised synthesis problem and can be solved using results of game theory [MAW89],[PR89],[Tho95] to determine a finite state winning strategy. That strategy can be implemented as a program. As in a distributed synthesis problem more than two players exist, this approach cannot be directly extended.

A formal model of distributed games is introduced by Mohalik and Walukiewicz [MW03], where n players are playing against an environment that has full information about the signals. Each player has a local view on the game. The distributed synthesis problem is equivalent to finding a distributed strategy that is a collection of local strategies which win against the environment. [MW03] gives two translations which allow to reduce the n -player-game successively into a 2-player-game. After that, strategy generation is possible. Due to the

properties of the simplification algorithms, the 2-player strategies can be decomposed to local strategies in the n -player game.

Finkbeiner and Schewe [FS05] introduced a criterion to classify decidable architectures. Jens Regenberg [Reg05] implemented the translations to produce pipeline architectures from architectures that are by the criterion of Finkbeiner and Schewe decidable. By this a larger class of architectures can be handled.

This thesis combines the mentioned formal approaches. It gives implementations of datastructures and algorithms that allow to describe and solve the distributed synthesis problem with the help of distributed games.

From the developed implementations *ReaSyn* was developed together with Jens Regenberg. *ReaSyn* is a tool to decide if a reactive system is realizable. In case of realizability it can synthesize the system. *ReaSyn* starts with definitions of an architecture and a specification. [Reg05] describes, how a pair of a specification and an architecture can be translated into games. The resulting games are transformed and solved by the methods presented in this thesis. The resulting strategies are translated into a valid implementation for each process by rebuilding the local strategies for each component of the n -player game. [Reg05] explains how this is done.

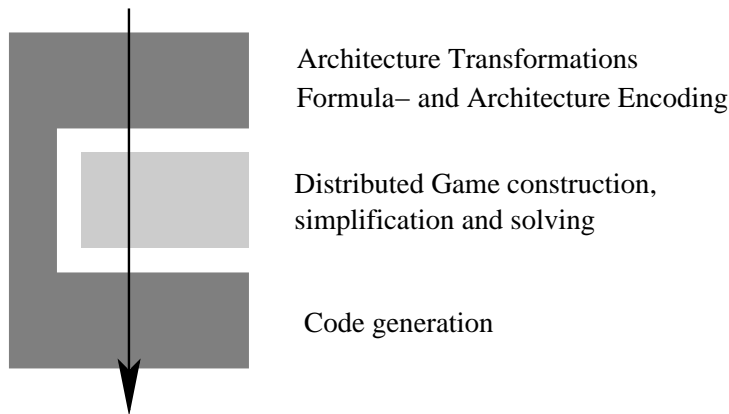


FIGURE 1.1 Interaction between the two theses

Figure 1.1 describes the interaction between the two theses. The information flows in a unidirectional way starting from the architecture and specification to be encoded into games. Then they are combined in one distributed game and successively solved. At the end, PROMELA code defining a valid implementation for each of the processes is generated.

Observe that this thesis is concerned with the core part depicted in the middle of figure 1.1. The organization of the work on the different parts and the proper definition of interfaces is an additional challenge to the thesis. As a common implementation environment is used for *ReaSyn*, the class definitions and data

structures are kept abstract and can so be used in both theses for different purposes.

The following section gives an overview on the content of each chapter.

Structure of this thesis

This thesis is divided into eight chapters. The following paragraphs provide a quick overview on the subsequent chapters.

Chapter 1, the present chapter, gives an introduction into the topic and relates the work to previous works.

In Chapter 2, *ReaSyn* is introduced. The different parts are mentioned and briefly described. It is pointed out, how the work was divided between this thesis and [Reg05].

Chapter 3 explains the theoretical concept of games and gives fundamental definitions which will be used in later sections. Determinacy of games is explained and a theorem that guarantees determinacy is presented. Based on the theorem, algorithms to find strategies are developed and proven.

The idea of games as defined in Chapter 3 helps to understand the distributed games introduced in Chapter 4. Transformations to reduce the number of players in a distributed game are explained. It is then shown that the winning situation is invariant under those transformations and that winning strategies can be carried over.

In Chapter 5 the distributed synthesis problem is introduced before the representations of architecture and specification are described. Next it is shown that the distributed synthesis problem is decidable for pipeline architectures.

Chapter 6 gives implementations building on the results of Chapters 3 to 5. This establishes the core part of *ReaSyn*. It is explained how the distributed game is constructed and how it can be simplified. The transformations from Chapter 4 are presented as algorithms.

Chapter 7 gives results of benchmark tests and the proceeding in the project.

The final chapter contains a summary of the several steps and insights throughout the work.

In the appendix brief explanations of the developed classes by UML diagrams and descriptions are given.

Chapter 2

ReaSyn

ReaSyn is a tool taking a setup of processes and a behaviour specification as input and synthesizes the whole system. This means it generates program code for each process such that the overall behaviour satisfies the specification.

ReaSyn was developed in C++ mainly for performance issues. The advantages of template datastructures and classes were used to create versatile implementations. The sourcecode and a precompiled executable is available under [MR].

This chapter provides additional information about the overall program. For thorough descriptions on how the program is invoked and available options, please refer to [Reg05]. For more information on the developed classes and function please take a look at the appendix of this thesis.

2.1 A closer look

ReaSyn consists of three parts. In figure 2.1 these parts are presented as discrete blocks named layer 1, 2 and 3.

The first and third part handle in- and outputs, respectively, whereas the middle part checks if the input is realizable and, if so, produces an implementation.

To give a brief overview of the tasks of each part, they are shortly discussed in the following. A deeper insight on the middle part is given in the subsequent chapters. For now it should be mentioned that the middle part works on games, a formal model in computer science that is defined in Chapter 3. The first and third part are contained in [Reg05].

- Layer 1 translates the input specification which is given as a temporal formula into games, one for each process. Additionally, the architecture

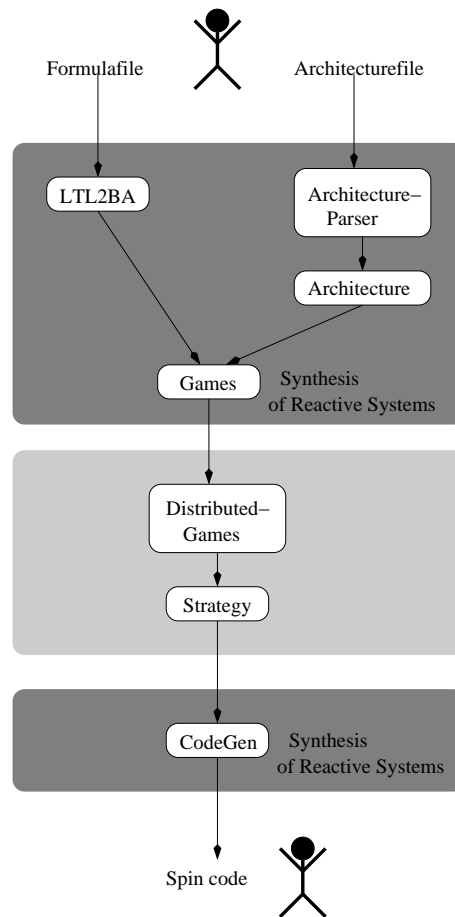


FIGURE 2.1 The partition of ReaSyn

that describes the process alignment and communication is encoded into a game.

- Next those games are merged into one big game, a so-called distributed game where the architecture plays against a hostile environment that is trying to violate the specification. A valid strategy can be viewed as a behaviour of the system, such that the architecture respects the specification independent of any external signals. A strategy is found by first reducing the distributed game to a two-player game and then applying known algorithms to solve the simplified game. The obtained non-deterministic strategies are passed to the next layer.
- From the generated n -deterministic strategy the third part of ReaSyn chooses a small and deterministic strategy, which is then translated into an implementation. That implementation is in PROMELA code, so that the resulting programs can be verified by SPIN [Hol03]. This was used as a validation opportunity in the development process of ReaSyn.

A detailed description on how *ReaSyn* works as well as a userguide can be found in Chapter 3 of Jens Regenbergs diploma thesis [Reg05].

2.2 Division of work

As *ReaSyn* was a project in cooperation with Jens Regenbergs; it was splitted to focus on the several interesting parts. Jens Regenbergs describes in [Reg05] the detailed proceeding concerning the first and third part (as fragmented in 2.1). This diploma thesis covers the details of the second part.

Chapter 3

Games

The main part of this thesis is concerned with games. To understand the functionality and power of games it is helpful to give a brief introduction on two-player games. In Chapter 4 distributed games will be introduced. Distributed games can be considered as a special kind of games and are better understood having the idea of commonly used games in mind.

Games in computer science are a theoretical construct, which is named “game” as it has some things in common with board games. In both games there are at least two players taking turns and every player is eager to win the party. What it means to win a game is later elaborately discussed. To get an idea of games, let us for now say a player would win, if the other player cannot move anymore. If one of the players has a special approach to force its adversary into such a position, independently of what the adversary is doing, the winner of the game is determined. Such an approach is called a strategy.

Bipartite games have the restriction that the players alternately take turns. For easier understanding it should be pointed out that only bipartite games are referred to throughout this thesis.

The following definitions introduce games and their fundamental attributes on a formal basis. Only two players are considered and the set of nodes owned by a player is denoted by P_σ , where $\sigma \in \{0, 1\}$, for player 0 and 1, respectively. This makes it easier, in case one talks about player σ to refer to the other player as $1 - \sigma$ or $\bar{\sigma}$. V stands for the set of all nodes $P_1 \cup P_2$.

3.1 Fundamental definitions

Definition 1: Infinite two-player games

An infinite two-player game

$$\mathcal{G} = (P_0, P_1, T, \chi)$$

consists of a finite set of nodes $V = (P_0 \uplus P_1)$ where each set $P_{\sigma \in \{0,1\}}$ belongs to one of the players. $T \subseteq ((P_0 \times P_1) \cup (P_1 \times P_0))$ is the set of directed transitions between game nodes. Additionally a game has a coloring function $\chi : P \rightarrow C$, that maps game nodes to colors. In this context colors are natural numbers ($C \subset \mathbb{N}$).

Definition 2: Play

A play in a game \mathcal{G} beginning at position q_0 is a possibly infinite sequence of nodes:

$$\pi : q_0 q_1 q_2 \cdots \in V^\omega \cup V^* V_0$$

where V_0 is the set of all nodes with exit degree 0. For all $q_i q_{i+1}$ there is $(q_i, q_{i+1}) \in T$.

Note, that in bipartite games the nodes q_i, q_{i+1} always belong to different players.

For plays, the coloring function as defined above can be extended in a straightforward way:

$$\chi(\pi) = \chi(q_0)\chi(q_1)\chi(q_2)\cdots$$

Definition 3: Initialized Game

A game \mathcal{G} is called initialized, if there is a set of starting nodes $S \subseteq V$ and for every play π the first node of the play $\pi(0)$ is included in S . In other words, only plays starting in S are valid plays. The game definition for initialized games extends to:

$$\mathcal{G} = (P_0, P_1, T, \chi, S)$$

To define the behavior of an infinite game, it is essential to come up with something finite. As the number of nodes of the games are finite, in an infinite play some nodes have to appear infinitely often. They can be specified by an infinity set.

Definition 4: Infinity set

The infinity set is the set of all nodes, that occur infinitely often in a play.

$$\text{Inf}(\pi) = \{q \in V : q \text{ occurs infinitely often in } \pi\}$$

It is still unclear what it means to win a play of a game. To define this, a winning set and a winning condition is introduced:

Definition 5: Winning Set

$W \subseteq V^\omega$ is the winning set, consisting of all infinite plays π that are accepted according to a appropriate winning condition.

In this diploma thesis the definition of a minimal parity winning condition is sufficient as [Reg05] already transformed the beforehand occurring winning conditions into minimal parity conditions. Please refer to Jens Regenbergs thesis [Reg05] on how this is accomplished.

Definition 6: Minimal parity winning condition

For all $\pi \in V^\omega$ the minimal parity winning condition defines the winning set W_σ of player $\sigma \in \{0, 1\}$ to be:

$$\pi \in W_\sigma \text{ iff } (\min(\text{Inf}(\chi(\pi))) \bmod 2) = \sigma.$$

The definition states, that if the minimum infinitely often occurring color is even, player 0 wins. As the minimal color can only be even or odd, if player 0 does not win, player 1 is obviously the winner. Formally this leads to:

Definition 7: The winner of a play

Player $\sigma \in \{0, 1\}$ wins a play of game \mathcal{G} , if

- π is a finite play $\pi = q_0q_1q_2 \dots q_l \in V^+$ and $q_l \in P_\sigma$ is a node, where player $(1 - \sigma)$ cannot move anymore. Such a position is called a dead end.
- π is an infinite play and $\pi \in W_\sigma$ (cf. Definition 6).

The question of interest is now how a player can win a play. As in a board game, he also needs a strategy here. A strategy defines, how a player behaves in a specific node by telling him where to go next. Such a decision can depend on previous observation, but in this thesis only memoryless strategies are used. Those strategies do not use a history or other information on what nodes were visited before.

Definition 8: Prefix of a play

The prefix of a play π is the first node in the sequence defined by π .

The set of all prefixes of a play is equivalent to the set of nodes occurring in a sequence.

Definition 9: Memoryless Strategy

A memoryless strategy for a player $\sigma \in \{0, 1\}$ is a function

$$f_\sigma : P_\sigma \rightarrow P_{(1-\sigma)}$$

that maps a σ -node to a $\bar{\sigma}$ -node. A play π respects f , if each of its prefixes respects f_σ . Let \mathcal{G} be an arbitrary game defined as above. The strategy f_σ is called winning or a winning strategy on $U \subseteq V$ if all plays which respect f_σ and start in a vertex from U are winning for player σ .

A player $\sigma \in \{0, 1\}$ is said to win a game \mathcal{G} on $U \subseteq V$ if he has a winning strategy on U .

The following example helps to better understand the defined attributes. In Figure 3.1 player 0 nodes are depicted as circles whereas player 1 nodes are squares. The number in a node denotes its color. A play of a game that is not initialized may start in any node. The player that owns the actual node decides where to go next (this is also often referred to as passing the token in one direction). This can then theoretically be repeated forever.

For every player a subset $U \subseteq V$ of game nodes can be defined, where the player can win a play in the following way:

Definition 10: Winning region

Given a game \mathcal{G} a winning region for player $\sigma \in \{0, 1\}$, denoted $W_\sigma(\mathcal{G})$ or in short W_σ is defined as the set U of all nodes n such that player σ has a memoryless winning strategy starting in a node of U .

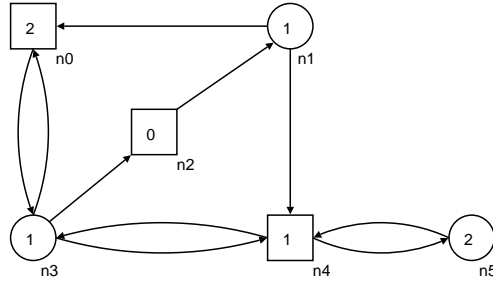


FIGURE 3.1 An example parity game

In order to find a winning region one can start with an initial region $U \subseteq V$, where a player is known to win. Then, that region can be enlarged by adding all nodes, where the player can force the play into U or the other player has no other choice than to move into U . The attractor set can also be defined over strategies:

Definition 11: Attractor set

The attractor set $Attr_{\sigma \in \{0,1\}}(\mathcal{G}, U)$ for player σ and set U is the set of nodes from which player σ has a memoryless strategy to attract the token to U or a dead end node of P_{σ} in a finite number of steps. The memoryless strategy is called $attr_{\sigma}(\mathcal{G}, U)$

In the example graph (cf. Figure 3.1) the attractor set $Attr_1(\mathcal{G}, \{n2\})$ is $\{n2\}$ as the only prior node $n3$ has other choices than to take the token into $n2$. $Attr_0(\mathcal{G}, \{n2\})$ is $\{n0, n1, n2, n3\}$

A part of a game can be called a subgame which is a game for itself, if the following definition preconditions hold:

Definition 12: Subgame

Let $S \subseteq V$ be any subset of V . The subgraph of \mathcal{G} induced by S is denoted $\mathcal{G}[S] = (P_0 \cap S, P_1 \cap S, T \cap (S \times S), \chi|_S)$. Where $\chi|_S$ is the restriction of χ to S .

The graph $\mathcal{G}[S]$ is a subgame of \mathcal{G} if every dead end in $\mathcal{G}[S]$ is also a dead end in \mathcal{G} .

In other words, in a subgame no new dead ends may be introduced.

$\mathcal{G}[\{n0, n1\}]$ in the example (cf. Figure 3.1) is no subgame, as it has a dead end whereas the original game has none. A subgame that follows the definition is for example $\mathcal{G}[\{n0, n1, n2, n3\}]$.

The two following definitions need to be understood for the upcoming strategy generation as it will be introduced in the following section. Both define subsets of game nodes according to the options of a player. A trap is a subset, where one player cannot escape. In a paradise, one player can always win. The following notation of traps and paradises includes the player that it is meant for as $\sigma \in \{0, 1\}$.

Definition 13: σ -trap

A σ -trap is a subset $U \subseteq V$ where player σ has no possibility to choose a successor outside of U . Additionally player $1 - \sigma$ is required to have a strategy that always chooses successors inside U .

Definition 14: σ -paradise

A σ -paradise in a game \mathcal{G} is a $\bar{\sigma}$ -trap U , such that in $\mathcal{G}[U]$, U is a σ -winning region.

Intuitively, a σ -paradise in a game is a $\bar{\sigma}$ -trap and σ wins from all nodes of this region using a memoryless strategy.

Considering the before mentioned example (cf. Figure 3.1) as a minimum parity game, $\{n4, n5\}$ is a 1-paradise. A 0-paradise can be obtained by choosing the nodeset $\{n0, n1, n2, n3\}$; a respective strategy for player 0 would be $f_0 := \{(n1, n0), (n3, n2)\}$. As by always taking those transitions, the play always repeats the cycle $(n0, n3, n2, n1)^\omega$, where the minimal, infinitely often occurring color is even (0). Thus player 0 is winning.

In the example $U = \{n4, n5\}$ is a 0-trap that happens to be winning for player 1, too. Observe that this is not required, as for example $U = \{n0, n3\}$ is a 1-trap, but obviously not winning for 0 (its minimum color is odd).

3.2 Determinacy

One important property of a game is whether one of the players can win a game no matter how the opponent acts. This property is called “determinacy”.

The winning regions and strategies for both players will later be computed by a recursive algorithm that is closely related to the following theorem’s proof.

To get an idea of how the algorithm works, it is first illustrated by an example. Consider Figure 3.2. It shows a two-player game \mathcal{G} with nodes a, b, c, d and colors 0,1. To compute the winning regions of \mathcal{G} , the algorithm starts with the

lowest color, namely 0. The player that would win if 0 appears infinitely often is player 0. Now the set of all nodes with color 0 is determined: $N = \{b, c\}$. Now the attractor set for player 0 to N is computed.

$$Attr_0(\mathcal{G}, \{b, c\}) = \{a, b, c\}$$

Now the remaining game Z_1 is $\mathcal{G} \setminus \{a, b, c\} = \{d\}$. As d is a dead end it is winning for player 1. This situation is shown on the left of figure 3.4. Now the attractor for player 1 to $\{d\}$ which is part of the winning region for player 1 is computed.

$$Attr_1(\mathcal{G}, Z_1) = \{c, d\}$$

Now the computed winning region is subtracted from the before calculated attractor set for 0:

$$Attr_1(\mathcal{G}, N) \setminus Attr_0(Z_1) = \{a, b\}$$

And for this set the algorithm determines the winning regions again. N is again the set of all nodes with minimal parity: $N = \{b\}$. And the attractor is determined:

$$Attr_0(\{a, b\}, \{b\}) = \{a, b\}$$

As the remaining Z is empty, the algorithm stops. The computed winning regions are the following:

$$WR_0 = Attr_0(\{a, b\}, \{b\}) = \{a, b\}$$

$$WR_1 = Attr_1(Z_1) = \{c, d\}$$

Figure 3.4 shows the resulting winning regions on the right.

For the general case, consider Figure 3.3. Suppose a game \mathcal{G} with colors $0 \dots k$ is given. The algorithm starts with the minimal occurring color. If this is k , player 0 can only win \mathcal{G} on dead end nodes in V , or on nodes from where he can force the token into such a dead end. Otherwise let $X_{\bar{\sigma}}$ be the afore defined $\bar{\sigma}$ -paradise (and a σ -trap). For $\sigma = 1 - (n \bmod 2)$ the set N is computed, having only nodes with color n . Then the attractor of N and the

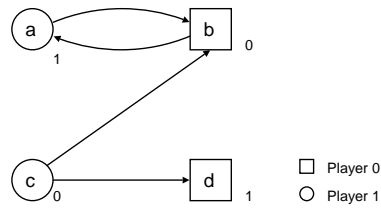


FIGURE 3.2 An example two-player game

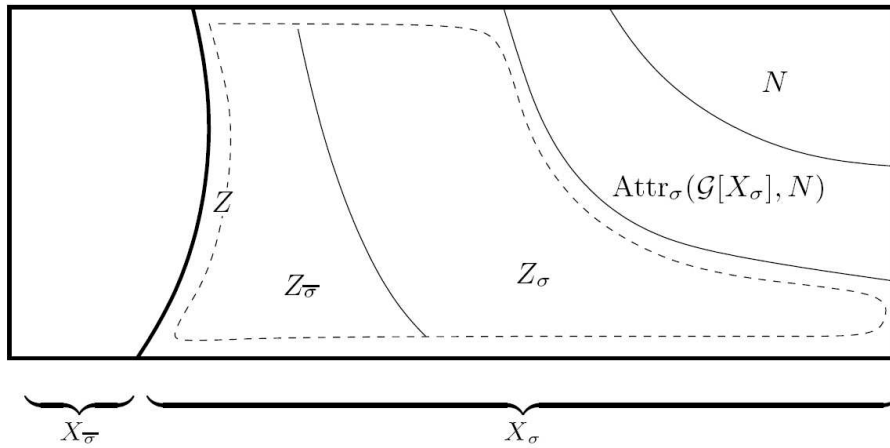


FIGURE 3.3 Construction of $X_{1-\sigma}$ and X_σ

winning regions for the subgame $\mathcal{G}[Z]$, where $Z = X_\sigma \setminus Attr_\sigma(\mathcal{G}[X_\sigma], N)$ are recursively computed. $X_{\bar{\sigma}}$ and $Z_{\bar{\sigma}}$ are united to extend the $\bar{\sigma}$ -paradise.

The afore mentioned example can be divided into the above mentioned subsets. Figure 3.4 shows the partition after the second step. Observe that Z is $\{d\}$ and afterwards no recursive steps are needed.

Theorem 1: Determinacy

The set of nodes of a parity game is partitioned into a σ -paradise and a $\bar{\sigma}$ -paradise

The proof of Theorem 1 is carried out over the number of states occurring in \mathcal{G} .

Proof of Theorem 1:

Base case ($n = 1$):

Since there is only one node in the game, the player owning the node has no successor and therefore loses the game.

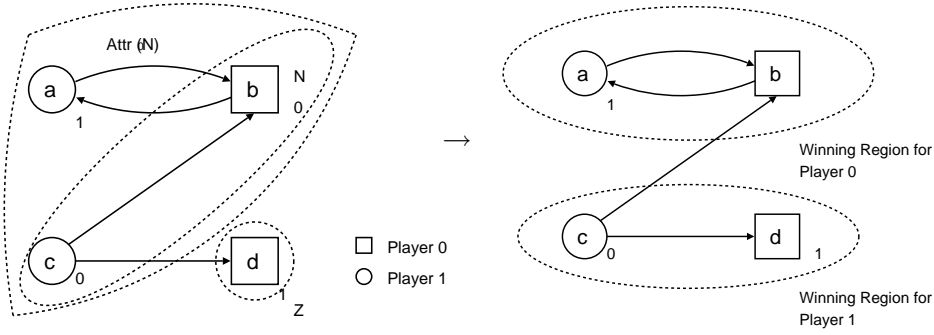


FIGURE 3.4 Winning regions algorithm applied on Figure 3.2

Induction step ($n \geq 1$):

Let j be the smallest color occurring on a node in V_σ . For $\sigma = (j \bmod 2)$ determine N , which is the set of all nodes which are colored by j . Build $Z = V_\sigma \setminus Attr_\sigma(\mathcal{G}, N)$. Observe that finding a winning strategy on $\mathcal{G}[Z]$ is a simpler problem, as it has less states than \mathcal{G} .

By induction hypothesis Z is partitioned into σ - and $\bar{\sigma}$ -paradises $Z_\sigma, Z_{\bar{\sigma}}$, respectively in $\mathcal{G}[Z]$ with the respective strategies z_σ and $z_{\bar{\sigma}}$.

Two cases must be differentiated:

- $Z_{\bar{\sigma}} = \emptyset$
in this case every position in \mathcal{G} is winning for σ with the following memoryless winning strategies:
 - in Z_σ follow the strategy in z_σ
 - in $Attr_\sigma(\mathcal{G}, N) \setminus N$ apply the attractor strategy $attr_\sigma(\mathcal{G}, N)$
 - in N any successor can be taken

Let π be a play conform with the above strategies starting at some node in X_σ . The three cases can occur. First, from some moment on, the token stays forever inside of Z and in this case some suffix of π is conform with z_σ and player σ wins. Second, the token is moved to a dead end in $V_{\bar{\sigma}} \cup Attr_\sigma(\mathcal{G}, N) \setminus N$, in which case σ wins as well. Third, the token visits infinitely often the maximal color in N and wins.

For the second case, if

- $Z_{\bar{\sigma}} \neq \emptyset$
 - $Attr_{\bar{\sigma}}(X_\sigma, Z_{\bar{\sigma}})$ is winning for $\bar{\sigma}$ in \mathcal{G} with $z_{\bar{\sigma}}$ in $Z_{\bar{\sigma}}$ and $attr_{\bar{\sigma}}(\mathcal{G}, Z_{\bar{\sigma}})$ in $Attr_{\bar{\sigma}}(\mathcal{G}, Z_{\bar{\sigma}})$.

$X_\sigma \setminus Attr_\sigma(X_\sigma, Z_{\bar{\sigma}})$ is a $\bar{\sigma}$ -trap and memoryless determined by induction hypothesis.

□

Wieslaw Zielonka's provides a more constructive proof in [Zie98] from which the later mentioned complexity is taken.

3.3 Finding a winning strategy

The straightforward constructive proof of Theorem 1 describes, how winning regions (σ -paradise and $\bar{\sigma}$ -paradise) can be computed. It can be summarized as followed:

Starting from a finite parity game \mathcal{G} , the first thing to do is to determine the lowest color n occurring in the game.

If this priority is k the base case holds: player $\bar{\sigma}$ can only win \mathcal{G} on dead ends or nodes from which he can force the token to such a dead end. Thus the $\bar{\sigma}$ -paradise is the set $Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)$ with $attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)$ as a memoryless winning strategy. And since $P \setminus Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)$ is a $\bar{\sigma}$ -trap and the minimum color of \mathcal{G} is k , it is easy to see that $P \setminus Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)$ is a σ -paradise.

Otherwise, for $\sigma = n \bmod 2$ ($n \neq k$), $w_{\bar{\sigma}}$ can be determined recursively. To find the regions it is sufficient to start with the lowest color. The above proof shows how by natural induction the winning sets are then extended by adding the found memoryless attractor strategies to the recursively computed memoryless strategies. The main argument for correctness is that the number of states is reduced in every recursion step.

3.4 Calculating winning regions

The before sections stated that it is possible to compute the winning regions of a finite parity game. The algorithm as introduced in [GTW02], runs on maximum parity games and is here adapted to a version that works on minimum parity games. The games considered are finite, and the highest possible color k is known.

3.4.1 Pseudo algorithm

Given a finite parity game, the algorithm **winning_regions** (cf. Figure 3.5) returns a tuple $((W_0, w_0), (W_1, w_1))$, where W_σ is the winning region and w_σ

```

winning_regions( $\mathcal{G}$ )

 $n := \min\{\chi(v) \mid v \in V\}$ 
 $\sigma := n \bmod 2$ 
If  $n = k$  then return  $((P \setminus Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset), w_{\sigma}), (Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset), attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)))$ 
//  $w_{\sigma}$  is some memoryless strategy for player  $\sigma$ 
// that avoids successor nodes within  $Attr_{\bar{\sigma}}(\mathcal{G}, \emptyset)$ 

// Otherwise
// compute  $W_{\bar{\sigma}}, w_{\bar{\sigma}}$ 

 $(W_{\bar{\sigma}}, w_{\bar{\sigma}}) := \mathbf{win\_opponent}(\mathcal{G}, \sigma, n)$ 

// compute  $W_{\sigma}, w_{\sigma}$ 
 $W_{\sigma} := V \setminus W_{\bar{\sigma}}$ 
 $N := \{v \in W_{\sigma} \mid n = \chi(v)\}$ 
 $Z := W_{\sigma} \setminus Attr_{\sigma}(\mathcal{G}[W_{\sigma}], N)$ 

 $((Z_0, z_0), (Z_1, z_1)) := \mathbf{winning\_regions}(\mathcal{G}[Z])$ 

 $\forall v \in W_{\sigma} \cap P_{\sigma} :$ 

 $w_{\sigma}(v) := \begin{cases} z_{\sigma}(v) & \text{if } v \in Z, \\ attr_{\sigma}(\mathcal{G}[W_{\sigma}], N)(v) & \text{if } v \in Attr_{\sigma}(\mathcal{G}[W_{\sigma}], N) \setminus N, \\ v' & \text{if } v \in N \text{ and } v' \in vE \cap W_{\sigma} \end{cases}$ 

return  $((W_0, w_0), (W_1, w_1))$ 

```

FIGURE 3.5 winning_regions

the respective memoryless strategy for player σ ($\sigma \in \{0, 1\}$). **winning_regions** uses a second function **win_opponent** (cf. Figure 3.6) that calculates a winning region and an according strategy for the opponent. The correctness of the proof follows directly from the constructional proof of Theorem 1. An attractor set can be computed in time $O(|V| + |T|)$, the running time of the winning region algorithm results in $O(|T| \cdot |V|^k)$ which is exponential in the number of colors and polynomial in the number of nodes.

Definition 15: Non-deterministic strategy

A non deterministic strategy for a player $\sigma \in \{0, 1\}$ is a function

$$f_{\sigma} : v_{\sigma} \rightarrow P_{\bar{\sigma}}$$

that maps a node of player σ to a set of winning successors.

```

win_opponent( $\mathcal{G}, \sigma, n$ )

( $W, w$ ) := ( $\emptyset, \emptyset$ )

Repeat
( $W', w'$ ) := ( $W, w$ )
 $X := Attr_{\bar{\sigma}}(\mathcal{G}, W)$ 
 $\forall v \in X \cap P_{\bar{\sigma}}$ :


$$x(v) = \begin{cases} w(v) & \text{if } v \in W, \\ attr_{\bar{\sigma}}(\mathcal{G}, W)(v) & \text{if } v \in X \setminus W. \end{cases}$$


 $Y = V \setminus X$ 
 $N := \{v \in Y \mid n = \chi(v)\}$ 
 $Z := Y \setminus Attr_{\sigma}(\mathcal{G}[Y], N)$ 

 $((Z_0, z_0), (Z_1, z_1)) = \mathbf{winning\_regions}(\mathcal{G}[Z])$ 

 $W := X \cup Z_{\bar{\sigma}}$ 
 $\forall v \in W \cap P_{\bar{\sigma}}$ :


$$w(v) = \begin{cases} x(v) & \text{if } v \in X, \\ z_{\bar{\sigma}}(v) & \text{if } v \in Z_{\bar{\sigma}}. \end{cases}$$


Until  $W' = W$ 
return ( $W, w$ )

```

FIGURE 3.6 win_opponent

To find winning strategies it is normally sufficient to choose winning successor nodes. The here calculated winning strategies are non-deterministic, as the algorithm does not choose successors but rather gives a list of all found successors. This gives more possibilities to the later applied code generation. It can look for in some way optimal paths within the tree spanned by the strategies.

The algorithm for computing the winning regions and strategies were implemented as a member function of the class Game (see Appendix).

Chapter 4

Distributed Games

This chapter introduces distributed games in general. They are later used to solve the distributed synthesis problem (cf. Chapter 5). The idea is, that one game has the role of a propositional formula and the other games stand for one processor each.

A distributed game can be considered as a model to simulate various simultaneously running games. From the view of a local game it is not possible to determine the global game position. This leads to general undecidability. Here a information hierarchy is used to ensure decidability. The hierarchy orders the processes according to their degree of information from the best to the worst informed. All reduction algorithms as used here respect and prevail the hierarchy.

The framework of distributed games as it is used here was presented by Mohalik and Walukiewicz [MW03]. Distributed games and their properties are introduced in the following. Some changes were applied to the model by [MW03]. As for example the input architectures are more specific, some properties are simplified. After the formal basis is established in this chapter, the next chapter deals with efficient construction and simplification of the formally defined distributed games.

4.1 Local Game

Local games behave like the games defined in Chapter 3 with the exception that they do not have winning conditions. For convenience the two players are here called player and environment with the respective sets of nodes P and E (rather than P_0 and P_1 cf. Chapter 3).

A short definition and some remarks on the properties of local games follow. For a deeper insight on the potentials and limitations of games please refer to

Chapter 3.

Definition 16: Local Game

A local game $\mathcal{G} = (P, E, T)$ is a bipartite game without winning condition, where P and E are player and environment nodes, respectively. The set of game nodes V is the disjunct union of player nodes P and environment nodes E . T is the transition function and $T \subseteq P \times E \cup E \times P$.

For $(v_1, v_2) \in T$, v_2 is called the successor of v_1 and v_1 the predecessor of v_2 . In a play environment and player take turns. According to the current node v , they choose a successor node $v' \in \{v'' \in V \mid (v, v'') \in T\}$.

4.2 Distributed Game

As stated before, distributed games are one model to simulate various concurrently running local games. Now follows a definition on distributed games the encoding will be explained later.

Definition 17: Distributed Game

Suppose $n + 1$ bipartite, **local games** are given $(\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \dots)$, a **distributed game** \mathcal{G} is also bipartite and it is defined as follows: $\mathcal{G} = (P, E, T, Acc \subseteq (E \cup P)^\omega \cup (E \cup P)^*)$, where:

- $P = E_0 \cup P_0 \times \dots \times E_n \cup P_n \setminus E$
- $E = E_0 \times \dots \times E_n$
- $T = T_{p(\text{layer})} \cup T_{e(\text{environment})}$.

The transitions for player and environment positions are different:

• **Player transitions:**

$$T_p \ni ((x_0, \dots, x_n) \mapsto (x'_0, \dots, x'_n))$$

$$\Leftrightarrow: (x_i \mapsto x'_i) \in T_i \text{ for all } x_i \in P_i \text{ and } x_i = x'_i \text{ for all } x_i \in E_i.$$

• **Environment transitions:**

$$E \times P \subseteq T_e \ni ((x_0, \dots, x_n) \mapsto (x'_0, \dots, x'_n))$$

$$\Rightarrow: (x_i \mapsto x'_i) \in T_i \text{ or } x_i = x'_i \text{ for all } x_i \in P_i \cup E_i.$$

In a distributed game a transition from a player to an environment position is only possible if there is a transition for all the local player positions and if all local environment positions stay the same.

Observe that, for a distributed player move, all positions have to change to local environment positions. Transitions on the environment side is a bit less restrictive, as for every position either the state stays the same or an arbitrary local transition can be taken. Note that at least one node has to change, as otherwise the global transition would lead into the same node which is not permitted because the distributed games are required to be bipartite.

Definition 18: Projection of a distributed game sequence

Let η be an $n + 1$ tuple, describing a distributed game node as mentioned above. With $l = 0, \dots, n$, let $\eta[l]$ denote the l -th component of η . For a sequence $\vec{v} = \eta_0[l]\eta_1[l] \dots$ denote the projection of the sequence to the l -th component.

Following the definition of distributed game transitions, a sequence projected to the i -local game, \vec{v} is of the form $e_0^+ p_0 e_1^+ p_1 \dots$

Definition 19: View of a process

Let \vec{v} be a play and $e_0^+ p_0 e_1^+ p_1 \dots$, its projection on the i -th component. The view of process i of \vec{v} is $view_i(\vec{v}) = e_0 p_0 e_1 p_1 \dots$

Definition 20: i -local strategy

An i -local strategy σ_i is a strategy in the local game \mathcal{G}_i .

Definition 21: Distributed strategy

A distributed strategy σ is a tuple of local strategies $\langle \sigma_0, \sigma_1, \dots, \sigma_n \rangle$. It defines a strategy in the distributed game \mathcal{G} by $\sigma(\vec{v}(x_1, \dots, x_n)) = (e_1, \dots, e_n)$, where $e_i = x_i$ if $x_i \in E_i$ and $e_i = \sigma_i(view_i(\vec{v} \cdot x_i))$ otherwise. σ is called the global strategy associated with the given distributed strategy.

By the definition of distributed games, any tuple of local strategies defines a global strategy, as it always suggests a valid move.

4.2.1 Examples

Figure 4.1 illustrates local games \mathcal{G}_0 and \mathcal{G}_1 , where player nodes are depicted as squares and environment nodes as round shapes. In the second game, player has fewer moves.



FIGURE 4.1 Two local games with rectangles as player and circles as environment nodes.

A distributed game \mathcal{G} of the local games $\mathcal{G}_0, \mathcal{G}_1$ is given in figure 4.2.

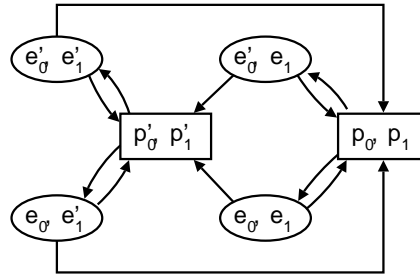
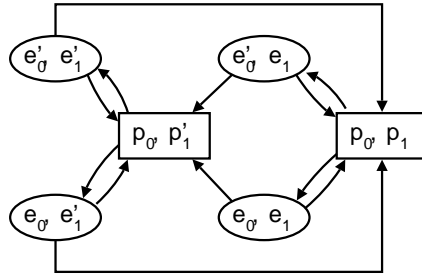


FIGURE 4.2 Distributed Game \mathcal{G} constructed from local games $\mathcal{G}_0, \mathcal{G}_1$

In this case the environment has less possibilities than it would have in a free product of \mathcal{G}_0 and \mathcal{G}_1 . For example in \mathcal{G}_0 the environment can move from e_0 to p'_0 and in \mathcal{G}_1 from e_1 to p'_1 , but there is no transition from $(e_0, e_1) \mapsto (p'_0, p'_1)$ in the distributed game.

Depending on what is modelled by the distributed game, this can make sense. If the winning condition states, that a winning game has to avoid environment positions with contemporary primed notions $((e_0, e'_1), (e'_0, e_1))$. The global winning strategy for player in the distributed game would then be: in position (p_0, p_1) always go to (e_0, e_1) and in (p'_0, p'_1) always to (e'_0, e'_1) . This strategy is also a distributed strategy: player 0 chooses e_0 in p_0 and e'_1 in p'_1 .

Changing the player position (p'_0, p'_1) to (p_0, p'_1) leads to the game as illustrated in figure 4.3. In this game a global winning strategy for player exists that in (p_0, p'_1) chooses (e'_0, e'_1) and in (p_0, p_1) chooses (e_0, e_1) as the next positions. But there is no distributed strategy because the local game 0 cannot determine from it's local view in which position the distributed game is. Suppose there is a distributed strategy which is winning from the vertex (e'_0, e_2) .

FIGURE 4.3 Distributed Game \mathcal{G} constructed from local games $\mathcal{G}_0, \mathcal{G}_1$

If environment moves to position (p_0, p_1') then player 0 should respond with e_0' . But if environment moves to (p_0, p_1) then the view of player 0 is the same and he will also move to e_0' , which is a losing move.

Hence there exist distributed games with global winning strategies for the players that do not have distributed winning strategies.

As in this thesis the local winning strategies are going to be derived from distributed strategies, the player needs the ability to determine the global position from its local view. This property is called environment determinism. The next section gives detailed information on how this is achieved. For now it is only important to see that environment determinism is required.

4.3 Transformations on Distributed Games

Theorem 1 in Chapter 3 states, that two player parity games are determined. Distributed games in general are not determined. In order to solve a distributed game it is first transformed into a two player parity game.

For a simplification of the distributed games, the following operations are introduced and their correctness is proven.

4.3.1 Division

The division operation reduces the number of local games. By multiple application of the division operation one can reduce a distributed game with an arbitrary number of local games to a distributed game with only two players. Each division operation reduces the number of local games by one.

In order to reduce the number of players, it is plausible that a player has to know the global game position from its local view. Therefore a form of deter-

minimism is required. Hence the following definitions are needed:

Definition 22: *i*-determinism

A game \mathcal{G} is *i*-deterministic if for every environment position η of \mathcal{G} and every transition $(\eta, \pi_1), (\eta, \pi_2) \in T_e$, if $\pi_1 \neq \pi_2$ then $\pi_1[i] \neq \pi_2[i]$.

A distributed game can be divided, if it is 0- and n -deterministic. The definition of the divided game is very intuitive if one considers the way distributed games are defined. The first and the last local game are merged together and every other local game stays the same. For the acceptance set ACC there is only a different notation used. The following two definitions are required to compare the rearranged tuples:

Definition 23: $flat : (V \times V) \times \dots \times V \rightarrow V \times V \times \dots \times V$

$$flat((x_0, x_n), x_1, \dots, x_{n-1}) = (x_0, x_1, \dots, x_n)$$

and analogously:

Definition 24: $flat^{-1} : V \times V \times \dots \times V \rightarrow (V \times V) \times \dots \times V$

$$flat^{-1}(x_0, x_1, \dots, x_n) = ((x_0, x_n), x_1, \dots, x_{n-1})$$

Division function

For the game $\mathcal{G} = (P, E, T, Acc)$, a function $DIVIDE(\mathcal{G})$ translates this game into a divided game $\tilde{\mathcal{G}} = (\tilde{P}, \tilde{E}, \tilde{T}, \tilde{Acc})$ with the following properties:

- $\tilde{P}_i = P_i$, $\tilde{E}_i = E_i$ and $\tilde{T}_i = T_i$ for $i = 1, \dots, n - 1$;
- $\tilde{E}_0 = E_0 \times E_n$ and $\tilde{P}_0 = (P_0 \cup E_0) \times (P_n \cup E_n) \setminus \tilde{E}_0$.
- For $(p_0, e_0) \in T_0$ and $(p_n, e_n) \in T_n$, we have the following transitions in \tilde{T}_0 :

$$\begin{aligned} (p_0, e_n) &\rightarrow (e_0, e_n), \\ (e_0, p_n) &\rightarrow (e_n, e_0), \\ (p_0, p_n) &\rightarrow (e_0, e_n). \end{aligned}$$

The global moves from environment positions and the acceptance set are defined using the above stated $flat$ function:

- $(\eta \rightarrow \pi) \in \tilde{T}$ if $(flat(\eta) \rightarrow flat(\pi)) \in T$

- $\widetilde{Acc} = \{\vec{v} : flat(\vec{v}) \in Acc\}$.

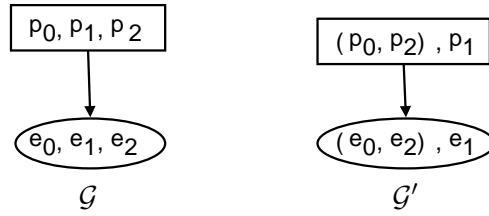


FIGURE 4.4 \mathcal{G}' is the result of $divide(\mathcal{G})$

The division function can easily be explained by an example: suppose a very simple game of three players is given (cf. figure 4.4). The possible distributed game player nodes are $((p_0, e_2), p_1)$, $((p_0, p_2), p_1)$ and $((e_0, p_2), p_1)$. $((e_0, e_2), e_1)$ is the respective environment node. As in the local games there were only transitions from (p_0, e_0) and (p_2, e_2) as depicted in figure 4.4 the resulting divided game has only one transition $((p_0, p_2), p_1) \rightarrow ((e_0, e_2), e_1)$. This result is shown as game \mathcal{G}' in figure 4.4.

Theorem 8 of [MW03] states that a given $n + 1$ distributed game with 0- and n -determinism has a distributed winning strategy **iff** there is one for the divided distributed game $\widetilde{\mathcal{G}}$. The theorem holds for every game position. It is the main argument for the correctness of the following division operation. A detailed proof of the theorem is given in [MW03] by Lemma 9 and 10.

Theorem 2:

Let \mathcal{G} be a 0-deterministic and n -deterministic distributed game of $n + 1$ players. For every position η of \mathcal{G} : there is a distributed winning strategy from η *iff* there is one from $flat^{-1}(\eta)$ in $\widetilde{\mathcal{G}}$.

4.3.2 Glue

As mentioned in the previous section 0- and n -determinism is an inevitable precondition for the division function. At the beginning, meaning before the first division, the distributed game is 0- and n -deterministic, by setup. This is because the specification game is determined and the n -th game has total information about the environments variables. Nondeterminism in one of the positions can arise after a divide step, as this step merges the first and last old game into one new local game on position 0.

To determinize a game, the gluing operation is introduced. Note that it only works if the following prerequisites hold.

Definition 25: A game \mathcal{G} is I-glueable if it satisfies the following conditions

- (i) \mathcal{G} is 0-deterministic
- (ii) \mathcal{G} has no 0-delays: if $(e_0, e_1, \dots, e_n) \rightarrow (x_0, x_1, \dots, x_n)$ then $x_0 \in P_0$
- (iii) The winning condition ACC is a parity condition on P_0 : there is a map $\Omega : (P_0 \cup E_0) \rightarrow \mathbb{N}$ such that $\vec{v} \in ACC$ **iff** $\liminf_{i \rightarrow \infty} \Omega(\text{view}_0(\vec{v}))$ is even.

The later construction of distributed games (cf. section 6.3) will ensure that properties (i), (ii) and (iii) always hold. Thus, the here used distributed games are always I-glueable. By Theorem 3 the I-glueable property is sufficient for the correctness.

In [MW03] there is also a property II-glueable introduced. A checking algorithm for the property was implemented in *ReaSyn* and may be used for later extensions. For the sake of completeness the II-glueable property is mentioned here.

Definition 26: A game \mathcal{G} is II-glueable if it satisfies the following conditions

- (i) The moves of other players are not influenced by P_0 moves: for every transition $(e_0, e_1, \dots, e_n) \rightarrow (x_0, x_1, \dots, x_n)$ and every other environment position e'_0 we have $(e'_0, e_1, \dots, e_n) \rightarrow (x'_0, x_1, \dots, x_n)$ for some x'_0 .
- (ii) The moves of player 0 are almost context independent: there is an equivalence relation $\sim \subseteq (E_0 \times P_0)^2$ such that, if $(e_0, e_1, \dots, e_n) \rightarrow (p_0, x_1, \dots, x_n)$ then for every $(e'_0, p'_0) : (e'_0, e_1, \dots, e_n) \rightarrow (p'_0, x_1, \dots, x_n)$ **iff** $(e'_0, p'_0) \sim (e_0, p_0)$.
- (iii) \mathcal{G} has no 0-delays (cf. I-glueable (ii)).
- (iv) The winning condition is a conjunction of the winning condition for players 1 to n and the condition for player 0. Additionally the condition for player 0 is a parity condition.

The definitions for the glueable characteristics are:

Glueing Operation

If a game is conform to one of the *glueable* preconditions (cf. 26), the following **glue** operation can be applied to achieve a determinized game (relative to the position 0). Then a further reduction of the number of players is possible (cf. 4.3.1).

In the following definition an abbreviated notation for (x_0, x_1, \dots, x_n) , namely (x_0, \bar{x}) is used.

For the game $\mathcal{G} = (P, E, T, Acc)$ with $n + 1$ players, a function $GLUE(\mathcal{G})$ translates this game into a glued $n + 1$ -player game $\tilde{\mathcal{G}} = (\tilde{P}, \tilde{E}, \tilde{T}, \tilde{Acc})$ with the following properties:

- $\tilde{P}_i = P_i, \tilde{E}_i = E_i$ and $\tilde{T}_i = T_i$ for all $i = 1, \dots, n$
- $\tilde{P}_0 = 2^{E_0 \times P_0}$ and $\tilde{E}_0 = 2^{P_0 \times E_0}$
- $\tilde{p} \rightarrow_0 \tilde{e}$ if, for every $(e, p) \in \tilde{p}$, there is $(p, e') \in \tilde{e} \cap T_0$
- $(\tilde{e}_0, \bar{e}) \rightarrow (\tilde{x}_0, \bar{x}) \in \tilde{T}$ for $x_0 \neq \emptyset$,
where $\tilde{x}_0 = \{(e_0, x_0) \mid \exists (p', e_0) \in \tilde{e}_0. (e_0, \bar{e}) \rightarrow (x_0, \bar{x})\}$

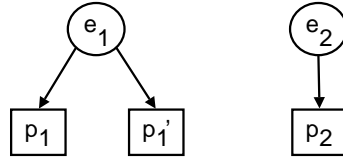


FIGURE 4.5 Two local games

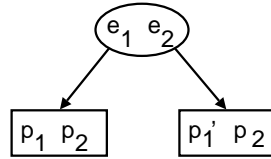


FIGURE 4.6 The Distributed Game

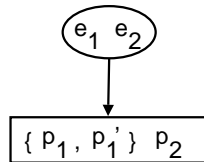


FIGURE 4.7 Glued Distributed Game

To define \tilde{Acc} , two auxiliary functions are needed:

The definition of distributed game transitions allows a local game to stay for several global moves in a local environment position. Note that a local player

position has to change. Therefore a projection of a play \vec{v} to the i -th component can be written as $e_0^+ p_0 e_1^+ p_1 \dots$ (where e_i^+ denotes the stuttering in position e_i).

Definition 27: $view()$

The view of a process i of $\vec{v} = e_0^+ p_0 e_1^+ p_1 \dots$ is $view_i(\vec{v}) = e_0 p_0 e_1 p_1 \dots$

Definition 28: $threads()$

A thread in $\vec{u} = u_1, \dots, u_{2k} \in (\widetilde{E}_0 \cdot \widetilde{P}_0)^+$ is any sequence $e_1 p_1 \dots e_k p_k \in (E_0 P_0)^+$ such that $(p_{i-1}, e_i) \in u_{2i-1}$ and $(e_i, p_i) \in u_{2i}$ for all $i = 1 \dots k$. Threads also work on infinite sequences. Let $threads(\vec{u})$ be the set of threads in \vec{u} .

With Definitions 27 and 28 the set of accepting sequences can be defined:

Definition 29: Accepting sequences

Let \vec{u} be a sequence of sets of pairs of nodes of the game. Then $\vec{u} \in \widetilde{Acc}$ **iff** every $\vec{v} \in threads(view_0(\vec{u}))$ satisfies the parity condition Ω .

The condition “every $\vec{v} \in threads(view_0(\vec{u}))$ ” satisfies the parity condition Ω ” is a regular winning condition. This means that there is a deterministic parity automaton recognizing sequences over $(\widetilde{E}_0 \cdot \widetilde{P}_0)$ with this property. By adding this automaton as a component for player 0 positions the glued game $\widetilde{\mathcal{G}}$ can be converted to a game with parity winning conditions for player 0.

For *II*-glueable games the definition additionally requires \vec{u} to satisfy the conditions for players $1, \dots, n$. But as before mentioned *I*-glueability is sufficient for the here presented distributed games.

Theorem 3:

Let \mathcal{G} be a *I*-glueable or *II*-glueable game. There is a distributed winning strategy from a position η in \mathcal{G} **iff** there is a distributed winning strategy from the position $\tilde{\eta}$ in $GLUE(\mathcal{G})$.

For easier provability of the above theorem, the two different glueable properties were introduced. The chain of proof differs from the two preconditions. Detailed proofs can be found in [MW03] in sections 5.1 (for *I*-glueable) and 5.2 (*II*-glueable).

Remarks

With the introduced distributed games it is possible to create a system that represents various local games. As the here used construction implements a information hierarchy by the used pipeline architectures, the preconditions for glue and divide are always satisfied. The divide operation allows to reduce the number of games, if the game meets the determinism precondition - otherwise the glue operation allows to establish determinism in the considered games.

At the end of the simplifications a two player game is gained.

Chapter 5

Distributed Synthesis Problem

A distributed system consists of a set of processes, that compute output values from given input histories. One interesting problem in formal computer science is if it is possible to synthesize a distributed system. A distributed system is defined by an architecture of processes with their communication signals and a logical formula. Synthesizing a distributed system is the task to find a implementation for each process such that the formula is satisfied.

This chapter defines the mentioned vocabulary formally. At the end of the chapter, a theorem is given, that states the decidability of the distributed synthesis problem for pipeline architectures. In other words it states that it is always possible to either synthesize a distributed system or to see that it is not realizable.

Distributed Synthesis is defined over processes that are entities with in- and output channels. A process reads from it's input channels and writes to its output channels.

An implementation is a description of a behaviour, that defines how to act upon a specific input history. In this context a processor acts according to an implementation, if the output is computed by running the implementation on the inputs. An architecture defines how processors are interconnected. In an architecture, every processor implements a deterministic function, that maps histories of input signals to output signals.

A distributed system consists of multiple processors that share some information over channels. They run in a usually common environment. The system specifies possible interactions between the processes and the environment by its structure.

Definition 30: Specification

A specification is a LTL [GO01] formula describing properties of a system.

A system is said to satisfy a specification, if the properties defined by the specification are fulfilled.

The **distributed synthesis problem (DSP)** is to find an implementation for each of the processors such that the overall behaviour of the system satisfies a given specification.

Definition 31: Distributed Synthesis Problem

Given an architecture \mathcal{A} and a LTL formula φ the Distributed Synthesis Problem (\mathcal{A}, φ) is to decide whether an implementation for \mathcal{A} satisfying φ exists. Additionally, if there is such an implementation, the problem extends to finding one.

Pnuelli and Rosner [PR90] showed that the problem of realizing a given propositional specification over a given architecture is undecidable in general. They also showed that it is decidable for the very restricted class of pipeline architectures. These results are based on the work of Peterson and Reif in [PR79] on games of incomplete information.

In [KV01] a decision procedure for two-way pipelines and one-way rings is given. The focus in this thesis lies on pipeline architectures. [FS05] describe a special property of architectures, called information fork: a criterion that characterizes all architectures for which distributed synthesis is undecidable. The transformations of [Reg05] can handle all fork-free architectures as input and produces architectures in which the information hierarchy is established. In those architectures the processes are ordered according to their degree of information. Every better informed process can simulate all subsequent processes as it has strictly more information. In a next step those architectures are translated into pipeline architectures. [Reg05] provides implementations of those algorithms.

The two fundamental concepts of an architecture which specifies the processes and their interaction, and specification that is to be respected are defined in the following two sections of this chapter. With that background and with the help of the before chapters, synthesis on the base of distributed games can be done with the algorithms suggested in the third section of this chapter.

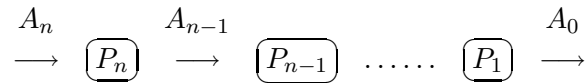


FIGURE 5.1 A pipeline architecture consisting of n processes. The in- and output signals are represented as labeled arrows.

5.1 Architectures

An architecture consists of processes acting in a specific environment. Processes in this thesis always communicate via channels through the environment. In other words: the signals are not hidden from the environment.

Practical examples on such architectures can be found in networking setups, where machines communicate over external channels with each other.

5.1.1 Pipeline Architectures

Definition 32: Pipeline Architecture

A pipeline architecture \mathcal{A} consists of a finite number of n processors P_1, \dots, P_n . Where there are disjoint alphabets A_{i-1} assigned to output of processor P_i the input alphabet of the pipeline is A_n .

The execution of a pipeline architecture follows in rounds, where the processor connected to the environment (P_n) gets the initial input over channel A_n . Now the next processor works on that input and so on. A round is finished, when the last process (P_1) produces its output.

Pipeline architectures are strictly ordered processes, that communicate via unidirectional channels. As soon as the implementation is fix, better informed processes can simulate those with a lower degree of information. The information has no delay.

In the model presented in this thesis, properties on signals are always checked at the end of a round.

5.1.2 Architecture transformations

Regenberg [Reg05] gives extensive explanations what kind of architectures can be transformed into pipeline architectures. In [Reg05] architecture transforma-

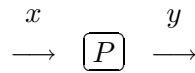


FIGURE 5.2 A pipeline architecture consisting of one process.

tion approaches are presented, which respect the fork criterion introduced by Finkbeiner, Schewe in [FS05] into pipeline architectures. Also transformations into games that form the interface to the in hand thesis are given.

5.1.3 Architecture game

Figure 5.2 shows an exemplary architecture consisting of a single process.

If the signals x, y are binary then P has all four possible functions $f_{i \in \{0,1,2,3\}} : \{0, 1\} \rightarrow \{0, 1\}$, that could be applied. A suitable game is given in figure 5.3, where the functions are denoted as $f_{i \in \{0,1,2,3\}}$.

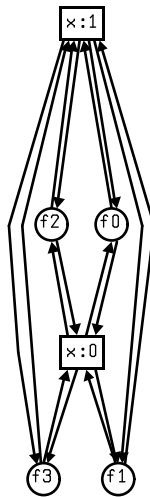


FIGURE 5.3 A process game. The squared player vertices contain variable assignments and the round environment vertices contain functions.

Observe that a player always chooses a function and the environment chooses the respective input.

5.2 Specification

ReaSyn covers LTL specifications directly or deterministic Büchi automata representing such specifications. Regenber [Reg05] describes the transfor-

mation into games. For a better intuition, LTL formulas are defined.

5.2.1 LTL

LTL or Linear Temporal Logic is a logical language that is used to specify computational behavior over time. LTL can be used to make a statement what happens to a set of variables in the future. For example one could state, that from a fixed but unknown point in the future for all later points a specific property is valid.

LTL defines properties on atomic propositions, that in this context are variables on signals, boolean atoms (true, false) over time. A LTL formula could for example state that from a point in the future something holds forever or that some other things never happen. Additionally, *ReaSyn* accepts simple arithmetic expressions on the atom level of LTL formulas.

Definition 33: *atom*

atom is the set of atomic propositions. It contains *true*, *false* and simple arithmetical expressions as defined by [Reg05].

Where in the above definition, *true* is the abbreviation for “ $1 == 1$ ” and *false* is “ $1 == 0$ ”.

Definition 34: Syntax of LTL Formulas [GO01]

The set of LTL formulas on the set *atom* of atomic Propositions is defined by

$$\begin{aligned} \varphi ::= & p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \\ & \mid X\varphi \mid \diamond\varphi \mid \square\varphi \mid \varphi_1 \cup \varphi_2 \end{aligned}$$

where $p \in atom$.

The arithmetic expressions relate signals of the architectures. Available operations are “+” for addition, “-” for subtraction, “*” for multiplication, “/” for integer division and “%” for modulo. The following self-explanatory relational operators are also allowed: “=”, “<”, “<=”, “>” and “>=”.

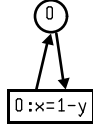


FIGURE 5.4 A simple specification game. Environment nodes are depicted as circles and player node as rectangles. The player node contains an identifier and an arithmetic expression, that is handled as an atomic proposition.

Definition 35: Semantics of LTL Formulas [GO01]

Let $\sigma = \sigma_0\sigma_1\dots$ be a word in Σ^ω with $\Sigma = 2^{atom}$ and φ a LTL formula. $atom$ is the set of atomic propositions as introduced in definition 33. The relation $\sigma \models \varphi$ (σ models φ) is defined as follows:

- $\sigma \models p \Leftrightarrow p \in \sigma_0$,
- $\sigma \models !\varphi \Leftrightarrow \sigma \not\models \varphi$,
- $\sigma \models \varphi_1 \vee \varphi_2 \Leftrightarrow \sigma \models \varphi_1$ or $\sigma \models \varphi_2$,
- $\sigma \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \sigma \models \varphi_1$ and $\sigma \models \varphi_2$,
- $\sigma \models X\varphi \Leftrightarrow \sigma_1\sigma_2\dots \models \varphi$,
- $\sigma \models \diamond\varphi \Leftrightarrow \exists k \geq 0. \sigma_k\sigma_{k+1}\dots \models \varphi$
- $\sigma \models \square\varphi \Leftrightarrow \forall k \geq 0. \sigma_k\sigma_{k+1}\dots \models \varphi$
- $\sigma \models \varphi_1 \cup \varphi_2 \Leftrightarrow \exists k \geq 0. \sigma_k\sigma_{k+1}\dots \models \varphi_2$
and $\forall 0 \leq i < k. \sigma_i\sigma_{i+1}\dots \models \varphi_1$.

To translate the LTL specification the fast LTL to Büchi automaton algorithms of Paul Gastin and Denis Oddoux [GO01] were used. As the resulting automaton may not be deterministic, it is determinized by applying Safra's construction [Saf88]. The resulting deterministic parity automaton is then translated into a parity game to fit the interface of this thesis. Up to that point the work of [Reg05] covers those topics.

In order to circumvent expensive translations (two exponential blowup in the size of the LTL formula) [Reg05] provides the possibility to give the specification directly as deterministic Büchi automaton.

5.2.2 Specification game

Consider the exemplary specification game in figure 5.4:

The specification represented by this game is $\square(x = 1 - y)$, where x and y are signal denominators. It means from now on and forever x is equal to $1 - y$.

The extra states in the game are obtained by the constructing and translating algorithms defined and explained by [Reg05].

Player nodes are depicted as circles and environment nodes as boxes. Every player node has a unique identification number. An environment node contains the expression and the name of the player node that lead to it. Note that the color of the nodes is not depicted in the figure. Nevertheless, there is a function mapping each node to a color.

5.3 Synthesis

5.3.1 Encoding

To use the distributed game framework to successfully decide the realizability of pairs of architectures and specifications as discussed before and synthesize an implementation, a proper encoding is needed. The definitions in this section were implemented and their implementation is described in chapter 6.

The distributed game is created from the local architectures and the specification in the following way: game \mathcal{G}_0 represents the specification and the local games $\mathcal{G}_1, \dots, \mathcal{G}_n$ take the role of the processes in the pipeline (process i corresponds to game i).

Specification game

The specification game is encoded from an underlying parity automaton that defines the specification. This is done extensively by Regenberg [Reg05], but in order to introduce the information a specification game node is carrying, the encoding is briefly explained.

Definition 36: Deterministic Parity Word Automaton

A deterministic parity word automaton A is a tuple $(Q, \Sigma, I, \delta, \mathcal{C})$ where Q is a finite set of states $\{q_0, \dots, q_n\}$, I indicates the set of starting states. Σ is the alphabet known to the automaton. $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function. And $\mathcal{C} : Q \rightarrow \mathbf{C}$ is a coloring function. \mathbf{C} is the set of colors ($\mathbf{C} \subset \mathbb{N}$).

In case of a Büchi automaton the chromatographic function changes to $\chi(q) = 0$ iff q is an accepted state and 1 otherwise.

Note that the winning condition of a parity automaton is equivalent to the one of a parity game (cf. section 3 definition 6).

The parity automaton is translated into a game in the following way. Let A be the automaton as defined above and $\mathcal{G} = (P_0, E_0, T_0)$ the game to be constructed and $\chi : P_0 \cup E_0 \rightarrow \mathbf{C}$ ($\mathbf{C} \subseteq \mathbb{N}$) the coloring function for the game nodes, then:

- $P_0 = Q \times \Sigma$
- $E_0 = Q$
- $(q, \sigma) \mapsto (q') \in T_0$ iff $\delta(q, \sigma) = q'$
- $(q) \mapsto (q, \sigma) \in T_0 \forall (q, \sigma) \in Q \times \Sigma$
- $T_0 = (P_0 \times E_0) \cup (E_0 \times P_0)$

The coloring function is defined as follows:

- $\chi : P_0 \rightarrow \mathbf{C}, \chi(v) = \mathcal{C}(v)$ for player nodes and
- $\chi : E_0 \rightarrow \mathbf{C}, \chi(v) = \mathcal{C}(v')$ if $v = (v', \sigma)$ for environment nodes.

The starting nodes of I are also starting nodes in the game.

From Theorem 3 and 4 of [Reg05] follows the correctness of this construction. A specification game has hence a winning strategy if and only if the specification is satisfiable.

Process game

The process games are simulating all possible behaviours of the processes. For this purpose, in each game the player chooses a function and the environment selects an input for the process. As the functions are encoded in environment and the input values in the player nodes, this is done by passing the token to the respective successor node.

For each component $i = 1 \dots n$ the process game of component i over the signals A_i is defined as $\mathcal{G}_i = (P_i, E_i, T_i)$, where:

- $P_i = A_i$
- $E_i = (A_i \rightarrow A_{i-1})$

and T_i is the complete set of transitions:

- $T_i = P_i \rightarrow E_i \cup E_i \rightarrow P_i.$

5.3.2 Decidability

For the decidability of distributed games some definitions are needed.

Let $\eta[1, i-1]$ denote the subsequence of the sequence η consisting of elements on positions from 1 to $i-1$; note that tuple η has also 0 position.

Definition 37: *i*-sequential

A distributed game \mathcal{G} is *i*-sequential if, for all environment positions η_1 and η_2 , the following holds: if π_1 is a successor of η_1 and π_2 is a successor of η_2 , $\eta_1[1, i] = \eta_2[1, i]$ and $\pi_1[1, i-1] \neq \pi_2[1, i-1]$ then $\pi_1[i] \neq \pi_2[i]$ and $\pi_1[i], \pi_2[i] \in P_i$.

Definition 38: $\langle 0, n \rangle$ -almost proper

A distributed game \mathcal{G} is called $\langle 0, n \rangle$ -almost proper if it satisfies the following:

\mathcal{G} is 0-deterministic

P_0 has no 0-delays

the winning condition is a parity condition on P_0

\mathcal{G} is *i*-sequential for all $i \in \{1, \dots, n\}$.

Definition 39: $\langle 0, n \rangle$ -proper

A distributed game \mathcal{G} is called $\langle 0, n \rangle$ -proper if it is $\langle 0, n \rangle$ -almost proper and also satisfies the following:

1. \mathcal{G} is *n*-deterministic

Observe that the condition (4) of the $\langle 0, n \rangle$ -almost proper criterion does not imply (1) of the $\langle 0, n \rangle$ -proper criterion, as the sequentiality does not say anything about player 0.

From the definitions of a pipeline game follows, that it is $\langle 0, n \rangle$ -proper. The following two lemmata are proven in [MW03] and will be used to proof the theorem on decidability later.

Lemma 1:

A $\langle 0, n \rangle$ -proper distributed game \mathcal{G} is dividable. Let \mathcal{G}' be the divided game, then \mathcal{G}' is a $\langle 0, n-1 \rangle$ -almost proper game.

Lemma 2:

A $\langle 0, n \rangle$ -almost proper distributed game \mathcal{G} is I -glueable and the glued game \mathcal{G}' is a $\langle 0, n \rangle$ -proper game.

Theorem 4: Decidability

The distributed synthesis problem is decidable for pipeline architectures.

Proof:

Given a pipeline game of $n + 1$ players, it is $\langle 0, n \rangle$ -proper by construction. Lemma 1 and 2 state, that the repetitive application of divide and glue result in a $\langle 0, n - 1 \rangle$ -proper game \mathcal{G}' . By Theorems 2 and 3, \mathcal{G} has a distributed winning strategy iff \mathcal{G}' has one.

By $n - 1$ repeated applications of divide and glue followed at the end by a divide operation, eventually a game consisting of one player against the environment evolves. This game has a parity winning condition. A distributed strategy in this game is just the winning strategy for the player. By Theorem 1, the existence of a winning strategy in this game is decidable. \square

Chapter 6

Getting to the point

The previous chapters state that distributed synthesis is decidable for pipeline architectures and proves the correctness of the here used algorithms. Now the algorithms and results of the previous sections are combined to build the core part of *ReaSyn*. While the focus of previous sections was on formal constructs, in this chapter the implementation and practical use comes to the fore.

6.1 A top down look on *ReaSyn*

ReaSyn starts with architecture and specification definition as presented in Chapters 5.1, 5.2. In a next step they are translated over various intermediate steps into games. As mentioned, this thesis concentrates on constructing and solving the distributed game. This section explains how this is established. The code generation is extensively explained by [Reg05].

6.2 Enriching the specification game

Information about the last occurring environment's choice of input signals are annotated to the player nodes which makes the resulting game dividable. The number of player nodes increases by this procedure.

Figure 6.1 shows a simple formula game (the formula used is $\Box("x = 1 - y")$). As the specification game on the left is constructed without connection to the external input variables of the environment, those are now added to every player node. The result of enriching the game can be seen on the right in figure 6.1.

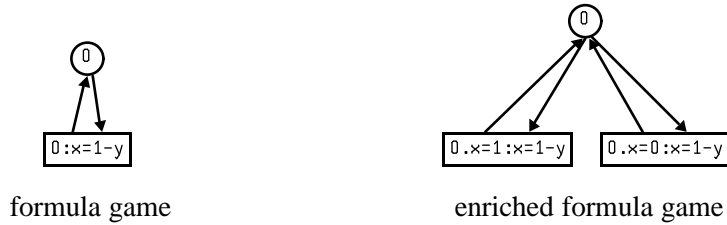


FIGURE 6.1 Enriching of the formula game as depicted on the left results in an enriched formula game on the right. The before explained player nodes is enriched by all possible environment values.

6.3 Constructing the Distributed Game

The construction starts with local games (as defined in 5.3.1) for every processor with empty winning conditions and one game for the specification, that comprises a parity winning condition.

Suppose n local games are to be encoded as a distributed game. Recall that the processor games will take positions $1, \dots, n - 1$ whereas the specification game is encoded in position 0.

For simplicity, the following notation is introduced. Let η be a node in a distributed game. Then $\eta(i)$ refers to the i -th component of the node, which stands for the position of the i -th local game. Furthermore \mathcal{G}_i means the i -th local game.

6.3.1 Building distributed game nodes

A formal specification of distributed game nodes is given in section 4.2. Now the practical construction is discussed. It turned out that the theoretical definitions are not very efficient if directly implemented, because they create lots of later unreachable nodes.

The presented construction only produces reachable nodes.

For this purpose, the execution begins from the start node of the distributed game, which is the cartesian product of all possible start nodes of the local games. From there, the reachable subgraph of the distributed game is computed.

Every distributed game node gets the parity of the node of \mathcal{G}_0 , which is coded in $\eta(0)$. An artificial dead end node is added to the distributed game. Owner of the node is the player. The dead end node contains “dead”-labels on every local position $\eta(i)$ ($i = 0 \dots n - 1$). The dead end’s parity has no account so

it can be set to an arbitrary value, let us say 1.

After this, a set of k start nodes is computed (plus the one dead end).

They are now memorized in a list of unprocessed nodes, where the successors are not yet computed. Now, beginning from that set, the new distributed game nodes are successively computed, where every new node is, if not previously seen, also added to the list of unprocessed nodes (provided that they are not already contained). Any new node is added, together with its incoming transition, to the distributed game.

For a distributed game node η the distributed successor nodes are determined as follows:

- if η is a player node:

S_i contain the successors of the i -th component of the i -th game. Let $(S_0, S_1, \dots, S_{n-1})$ be the sets of the local successors. Then all possible cartesian products $S = S_0 \times S_1 \times \dots \times S_{n-1}$ are computed and then added together with their edges $\{\eta\} \times S$ to the distributed game.

- if η is an environment node:

For every input value the functions of the environment nodes are evaluated and the local successors are added. Let $(S_0, S_1, \dots, S_{n-1})$ be the sets of the local successors computed as for player nodes. As an environment node contains functions for all components, it is required to check, whether they agree on the node description. Secondly it has to be tested if the evaluated expression is valid. Therefore, functions are evaluated with all startvalues coded in the nodes. The obtained variable assignment is then used to evaluate the possible successor node. If both succeeds, the node is a valid successor and is, together with the transition, added to the distributed game. The color of the newly added node is equal to the color of its 0-th component.

For an example how a successor of a environment node is determined, consider Figure 6.2, where f_2 defines the following function: $f_2 : \{0, 1\} \rightarrow \{0, 1\}$, $f_2(0) = 0$ and $f_2(1) = 0$. To check, whether $\langle 0.x = 0 : x = 1 - y, x : 0 \rangle$ is a valid successor of $\langle 0, f_2 \rangle$, $x = 0$ as defined by the player node is used as input for f_2 . By the definition of f_2 , y is 0. Now, with $y = 0$ and $x = 0$, $x = 1 - y$ is not valid. Therefore $\langle 0.x = 0 : x = 1 - y, x : 0 \rangle$ is a no valid successor of $\langle 0, f_2 \rangle$.

6.4 Simplifying the Distributed Game

The simplification of the presented distributed game is structured in two sections. One, in which the goal is to reduce the size of the graph, and a second, through which the reduction of local players is obtained. This is important for later synthesis.

6.4.1 Dead end removal

As the introduced nodes contain all information which is later used for synthesis and code generation by Jens Regenber [\[Reg05\]](#), special care with selecting the nodes which can be removed has to be taken.

One in this way save attempt is to remove all player nodes which have no choice but to take a successor that leads into a dead end. Recall that in a game the environment wins if it can force a token to a player dead end node. Also, all environment nodes that can force the token to such a node can be deleted, as this would be a possible winning strategy for them.

This is called dead end removal. It can be defined by the use of the attractor definition from [Definition 11](#) in the following way:

Definition 40: Dead end attractor

For a distributed game \mathcal{G} the dead end attractor is:

$$DeadEndAttr(\mathcal{G}) := Attr_{environment}(\mathcal{G}, \emptyset)$$

The correctness of the approach follows directly from the definition of the attractor set.

[Figure 6.2](#) shows a distributed game and [figure 6.3](#) shows the dead end free version. Observe that the number of game nodes is significantly reduced. The functions f_0 , f_1 , f_2 and f_3 represent the four possible total functions from $\{0, 1\}$ to $\{0, 1\}$. The commata mark the local parts of the distributed game nodes. Observe that the function that is not reduced is exactly the one that respects the specification.

In *ReaSyn* the *dead end*-optimization can be switched on with the commandline parameter *-optimize1*. It is run directly after constructing the distributed game. As no new dead end nodes are introduced by the subsequent operations (*glue* and *divide*) a later run would have no effect. [Chapter 7](#) shows some runs of *ReaSyn* with and without dead end optimization.

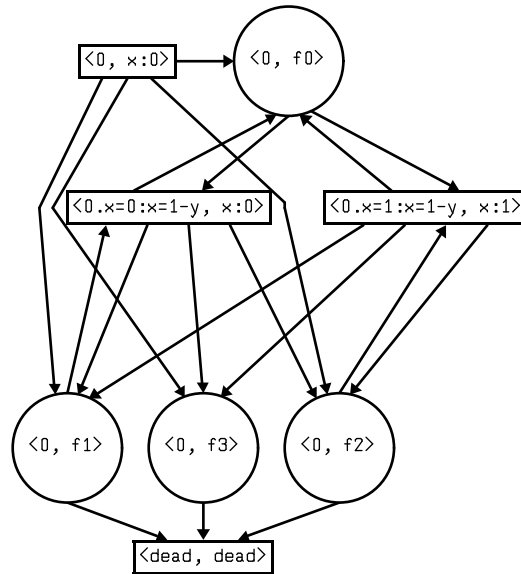


FIGURE 6.2 A distributed game after construction. The four functions f_i map bool values on bool values. $f_0(x) = \bar{x}$, $f_1(x) = 1$, $f_2(x) = 0$ and f_3 is the identity.

6.4.2 Removing environment winning positions

All environment winning positions after construction of the distributed games will later also be losing for player and can be removed. This is done by directly solving the distributed game and then deleting all nodes that are in the winning region of the environment. Note, that also all edges have to be removed as the subsequent algorithms explore the game graph.

The algorithm that removes all environment winning positions is available as member function “*EnvWin*” of class *Game*.

The solving algorithm is used as described in 3.4. As the *Deadend*-optimization is much faster because it has to determine the dead end attractor, it is in practice run before *EnvWin*. Despite to the *Deadend*-optimization, the *EnvWin*-algorithm has a high effect on distributed synthesis problems that have no solution. Those games have many game nodes which are winning for the environment.

In *ReaSyn* the *EnvWin*-optimization can be switched on by the commandline parameter *-optimize2*. It is run after the dead end removal and also after every glue operation.

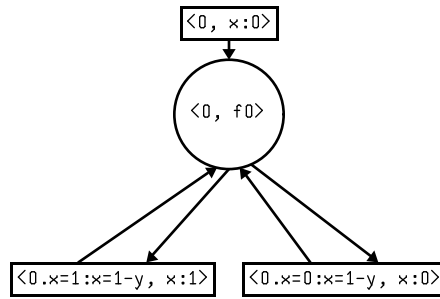


FIGURE 6.3 Result of dead end removal of the game in figure 6.2. All functions that are not valid for the given specification disappeared.

6.4.3 Colorspace reduction

The following algorithm reduces the number of colors occurring in a game. It first checks, if only two colors occur. In this case optimization is not required. Otherwise, the algorithm determines which colors are occurring. An array $occ[]$ of the size of the maximal occurring color + 1 marks every occurring color; in other words color j occurs iff $occ[j] = 1$.

Next the algorithm loops over the colors, starting with 0. For every slot in occ where a color is not used, the next occurring color is copied in that slot. The algorithm runs until the number of colors cannot be further reduced.

6.5 Reducing the number of local players

The player reduction is done by dividing as defined in section 4.3.1. If a game is not dividable it is glued in advance (cf. section 4.3.2). The glueing and dividing algorithms were implemented closely to their definitions. They are available as memberfunctions of class distributed game (cf. appendix section A.13).

All algorithms presented in the following are simplified for better readability. It is neglected how the datastructures are concretely accessed. The appendix gives more information on the available datastructures. With the following pseudo code and the source's inline comments the source files can be easily understood.

Before $divide()$ can be executed, the game has to be checked for 0 and n -determinism. If it is not deterministic, $glue()$ has to be executed first. Otherwise the game can be directly divided. Observe that, as the specification game and the first process game are deterministic (i.e. the distributed game is 0-deterministic), the first distributed game is always directly dividable. Recall

```
divideNode ( $x_0, \dots, x_n$ ) {  
    return ( $\langle x_0, x_n \rangle, x_1, \dots, x_{n-1}$ )  
}
```

FIGURE 6.4 DivideNode can be applied on a distributed game node

that the specification games determinacy comes from the determinizations of [Reg05]. The outermost process is deterministic because he has full information about the environment variables.

6.5.1 Divide

Divide uses a subfunction *divideNode* that rewrites a node according to the definition of *divide* (cf. Section 4.3.1).

The division algorithm is depicted in figure 6.5. It loops over the lists of player nodes *PlayerNodes*, environment nodes *EnvironmentNodes* and start nodes *StartNodes*. Every node is divided according to *DivideNode* (cf. Figure 6.4) and the distributed game is updated by the *addNew**-functions. For the player and environment nodes lists the transitions are adapted. After the division the distributed game ends up with n local players. The old players 0 and n are encoded as a pair in position 0.

If division is not directly possible because the game is not n -deterministic, it has to be determinized by *glue* in advance.

The running time of divide is linear in the number of all distributed game nodes, as it simply rewrites the tuples.

6.5.2 Glue

Figure 6.6 shows the main loop of *glue*. It starts with the start nodes of the distributed game and the subfunctions update a list *unseen* of unprocessed nodes. The algorithm stops, if *unseen* is empty. Successors for player and environment nodes are derived differently, the functionality is here divided in two subfunctions *pSucc* and *eSucc* for player and environment nodes respectively.

For player nodes, the successors are computed in two steps. First, all possible successors are computed and sorted by their tail (y_1, \dots, y_n) , meaning they have the same local view, put in the list of (*succs*) with the right tail. Second, for every list of *succs* every $\langle e_k, p_k \rangle$ of *set* a new pair node $\langle p_k, e'_i \rangle$ is added

```

divide() {
// create a new distributed game of n local games
// with positions  $1 \dots n - 1$  as before;
// calculate position 0 as follows

Foreach pNode of PlayerNodes {
  addNewPlayerNode(divideNode(pNode))
  Foreach successor of pNode {
    addNewTransition(divideNode(pNode), divideNode(successor))
  }
}
Foreach eNode of EnvironmentNodes {
  addNewEnvironmentNode(divideNode(eNode))
  Foreach successor of eNode {
    addNewTransition(divideNode(eNode), divideNode(successor))
  }
}
Foreach sNode of StartNodes {
  addNewStartNode(divideNode(sNode))
}
}
}

```

FIGURE 6.5 Divide reduces the number of local games

```

glue() {
for every startnode  $(x_0, x_1, \dots, x_n)$ 
add  $(\{< \text{---}, x_0 >\}, x_1, \dots, x_n)$  to unseen

While unseen is not empty {
let dNode be the first node in unseen

If dNode is a player node then
      pSucc(dNode)
else
      eSucc(dNode, dNode)
} }

```

FIGURE 6.6 *glue* loops over all unprocessed nodes and calls two subfunctions

to *rest*, where e'_i comes from the respective node of *succs* and *rest* is added to *possibleSuccs*.

Now a cartesian product construction of *possibleSuccs* gives the new first component of the distributed game nodes.

Since the possible successor nodes for the environment nodes are not required to be equal in y_1, \dots, y_n , the procedure is a bit different. Additionally, the 0-th position contains a set of pair nodes.

For performance issues, the colors of the glued nodes are by default approximated, as can be seen in Figures 6.7,6.8. This saves the expensive automaton construction as suggested by [MW03] and the required determinization which takes exponential time [Reg05]. If a system is considered not synthesizable by *ReaSyn* it should also be checked with the provided over-approximation. The over-approximation chooses the highest even, or if there are only odd number, the lowest odd number.

Nevertheless, [Reg05] describes how the automaton construction according to Definition 29 in Section 4.3.2 is implemented and gives the possibility to run it optionally. All implementations found by the implemented algorithms are correct. Even if there is a theoretical possibility that for a problem no result is found, throughout wide tests with *ReaSyn* the used under-approximation did not lead to any problems.

The running time for *glue* is n -exponential for n is the number of distributed game nodes, by [MW03]. As this the complexity class of a determinization algorithm, this is no surprise.

6.6 Solving the reduced game

After reducing the distributed game to a two player game, it is possible to solve the game. The algorithm introduced in section 3.4.1 is used to generate the winning regions and memoryless strategies $f_\sigma : P_\sigma \rightarrow P_{\bar{\sigma}}$ for the players $\sigma \in \{0, 1\}$ are produced.

If there is no such strategy, the formula is not satisfiable on the given architecture (cf. 4.3).

If and only if there is a winning strategy from all start nodes, the formula is satisfiable. The distributed game class produces strategies for player and environment. Instead of adding an arbitrary successor all possible successors are added in each step. By this, the possible paths through strategies increases resulting in more possibilities for the code generation. [Reg05] shows, that the so generated strategies can be used to synthesize the specification by evaluating

the nodes' information.

The code generation algorithms as implemented by [Reg05] can be used to generate program code, that is a valid implementation of the specification on the architecture.

```

pSucc(dNode){
// dNode is of the form ( $\langle e_0, p_0 \rangle, \langle e_1, p_1 \rangle, \dots \rangle, x_1, \dots, x_n$ )
dSNode := (NULL,  $x_1, \dots, x_n$ )
set :=  $\{ \langle e_0, p_0 \rangle, \langle e_1, p_1 \rangle, \dots \}$ 
initialize succs as a list of lists of pairs of nodes ( $\{ \{ \langle \_, \_ \rangle, \dots \}, \dots \}$ ).

Foreach  $\langle e_i, p_i \rangle$  of set {

    dSucc = getSuccessors( $(p_i, x_1, \dots, x_n)$ ) //break if empty
    Foreach  $(e'_i, y_1, \dots, y_n)$  of dSucc {

        add  $\langle (e'_i, y_1, \dots, y_n), \langle e_i, p_i \rangle \rangle$  to the list in succs containing
            all pairs with the same  $y_1, \dots, y_n$ 

    }
}

Foreach a :=  $\langle (e'_0, y_1, \dots, y_n), \langle e_0, p_0 \rangle \rangle, \langle (e'_1, y_1, \dots, y_n), \langle e_1, p_1 \rangle \rangle, \dots$ 
    of succs {

        added = false
        initialize sta as an empty list
        Foreach b :=  $\langle e_k, p_k \rangle$  of set {

            add each  $\langle p_k, e'_i \rangle$  to rest where b equals second part of a

            If rest is not empty then

                push rest on possibleSuccs; added := true

        }

        If added then {
            succSet is the full cartesian product of possibleSuccs
            Foreach s of succSet{

                add environment node  $(s, y_1, \dots, y_n)$  to the distributed game

                set the nodes color to smallest odd or highest even occurring in s

                add a transition  $dNode \rightarrow (s, y_1, \dots, y_n)$ 

                update unseen

            }
        }
    }
}

```

FIGURE 6.7 *pSucc* determines all successors of a player node

```

eSucc(dNode) {
// dNode is of the form ( $\{\langle p_0, e_0 \rangle, \langle p_1, e_1 \rangle, \dots \}, x_1, \dots, x_n$ )
dSNode := (NULL,  $x_1, \dots, x_n$ )
set :=  $\{\langle p_0, e_0 \rangle, \langle p_1, e_1 \rangle, \dots \}$ 
initialize succs as a list of lists of pairs ( $\{\{\langle -, - \rangle, \dots \}, \dots \}$ ).

Foreach  $\langle p_i, e_i \rangle$  of set {

    dSucc = getSuccessors( $(e_i, x_1, \dots, x_n)$ ) //break if empty
    Foreach  $(p'_i, y_1, \dots, y_n)$  of dSucc {

        add  $\langle (p'_i, y_1, \dots, y_n), \langle p_i, e_i \rangle \rangle$  to the list in succs containing
            all pairs with the same  $y_1, \dots, y_n$ 

    }
}

Foreach  $a := \langle (p'_0, y_1, \dots, y_n), \langle p_0, e_0 \rangle \rangle, \langle (p'_1, y_1, \dots, y_n), \langle p_1, e_1 \rangle \rangle, \dots$ 
    of succs {
    Foreach  $\langle (p'_i, y_1, \dots, y_n), \langle p_i, e_i \rangle \rangle$  of a {

        add  $\langle e_i, p'_i \rangle$  to succSet

    }

    add player node  $(succSet, y_1, \dots, y_n)$  to the distributed game
    set the nodes color to smallest odd or highest even occuring in succSet
    add a transition  $dNode \rightarrow (succSet, y_1, \dots, y_n)$ 
    update unseen

} }

```

FIGURE 6.8 *eSucc* determines the successors of a environment node

Chapter 7

Results

The sourcecode of *ReaSyn* and an executable are freely available on the project's homepage [MR]. A description of the user interface is available in Chapter 3 of [Reg05].

To demonstrate the efficiency of the program some examples are shown now and the size of the games after the various steps are listed in the tables below. All examples were run on a machine with a two gigahertz Intel pentium 4 processor and 512 Megabytes ram. To circumvent the definition of the syntax, the architecture and formulas are given as a graph and LTL formula, respectively. As *ReaSyn* is the first program for distributed synthesis, there is no program to compare it to.

7.1 Example 1

Consider the architecture in figure 7.1. It is a pipeline architecture of two processes. The specification is $\square(x = z)$ which requires the processes to transport the environment signal (x) through the pipeline. As the signal between the two processes (y) can have values of $\{2, 3\}$, the implementation has to deliver a proper encoding.

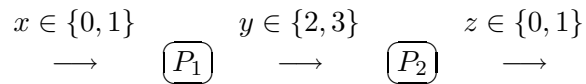


FIGURE 7.1 2-process pipeline consisting of two processes and three signals. The possible values of the signals are annotated as sets.

Table 7.4 shows some test results. Note that the division is left out as it does not

add nodes to the distributed game. The number of nodes is during optimization decreased by seven. The computation time without optimization is 110 ms, with optimization the computation takes 40 ms.

	Number of nodes
distributed game	22
dto. glued	-
running time	110 ms
optimized distributed game	15
dto. glued	-
running time	40 ms

TABLE 7.1 Results of $\square(x = y)$

As the game is deterministic glueing is not needed. *ReaSyn* returns an implementation for the processes where the first process maps 0 to 3 and 1 to 2. By the second process the values are translated according to the specification: 3 to 0 and 2 to 1. This is only one possible implementation. *ReaSyn* always tries to find small implementations, please refer to [Reg05] how this is done. The produced PROMELA code can be seen in Figure 7.2. It simulates the three processes. The code can be used to do a stepwise simulation in SPIN.

Let us now change specification $\square(x \rightarrow \diamond y)$, which means that whenever x is valid (has value 1) then eventually y is valid. The computation result can be seen in Table 7.2. In this case the optimization could not reduce the number of nodes. The running times are 24.76s without and 24.93s with optimization. But even with unsuccessful optimization the running time is not much higher.

	Number of nodes
distributed game	82
dto. glued	443
running time	24.76s
optimized distributed game	82
dto. glued	443
running time	24.93s

TABLE 7.2 Results of $\square(x \rightarrow \diamond y)$

The resulting implementation is the same as the one of the previous specification.

```
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!3;
        state = 0;
    :: ((x == 1)) ->
        yChan!2;
        state = 0;
    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 == 2)) ->
        zChan!1;
        state = 0;
    :: ((y == 3)) ->
        zChan!0;
        state = 0;
    fi;
od
}
```

FIGURE 7.2 A implementation for the given architecture and specification in PROMELA

7.2 Example 2

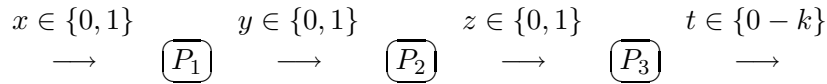


FIGURE 7.3 A_k , a 3-process pipeline consisting of three processes and four signals. The possible values of the signals are annotated as sets. $\{0 - k\}$ is an abbreviation for $\{j | 0 \leq j \leq k\}$.

This generic example shows the power of the optimization. $\square(y)$ is used as specification in order to keep the effect on the specification game construction low. In each run the number of possible values for signal t is increased. A_k indicates that the architecture as defined in Figure 7.3 has $i + k$ possible values for t (x, y and z are in $\{0, 1\}$). As the size of a processor game is in $O(|output|^{input})$ this significantly increases the size of the distributed game.

The results in Table 7.3 show the efficiency of the node reduction. As the underlying datastructures properly destruct unused nodes, the size reduction has a direct impact on the memory consumption of *ReaSyn*. Note, that for A_{15} the computation is not finished after 20 hours without optimization turned on and the memory consumption exceeded the available 512 MB. The optimized run had a memory peak of 250 MB and finished after 18 hours.

Number of nodes	A_1	A_5	A_{10}	A_{15}
distributed game	70	582	1942	7062
dto. glued	467	4051	13571	49411
running time	1:27.48	34:21.64	5:46:47.11	*
optim. dist. game	21	149	489	1763
dto. glued	34	291	971	3528
running time	0:03.23	1:40.81	32:29.16	**

TABLE 7.3 Results of $\square(y)$: the format of the running time annotation is *hours* : *minutes* : *seconds* . *milliseconds*. * was aborted after the memory exceeded. ** returned a valid implementation after 18 hours.

7.3 Example 3

The running times of the previous examples were influenced by the dead end optimization. To see what the precalculation of the environment winning states in the distributed game does, the following example is introduced.

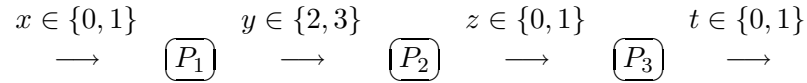


FIGURE 7.4 3-process pipeline consisting of three processes and four signals.

Again, a three process pipeline architecture is used (Figure 7.4 shows the setup). All signals are binary. The specification is $\Box(y) \wedge \Diamond(y = 0)$, which is clearly unsatisfiable, as if y has to be always true, it cannot turn eventually 0.

	Number of nodes
distributed game	138
dto. glued	860
optimized distributed game 1	41
optimized distributed game 2	0
dto. glued	-

TABLE 7.4 Results of $\Box(y) \wedge \Diamond(y = 0)$. The first optimization is done by dead end reduction and the second by solving the distributed game and deleting all environment winning nodes.

In this example, the dead end removal optimization deletes all nodes of the dead end attractor (cf. Chapter 6.4.1). By solving the resulting distributed game and deleting all nodes that are winning for the environment (according to Chapter 6.4.2) no nodes remain. Thus, the game cannot be won by the player and there is no valid implementation. This calculation takes 60 milliseconds while the unoptimized calculation lasts more than fourteen minutes. Observe that the unoptimized run needs one glue function. To gain a reliable information on the synthesizability the under-approximation was switched on for Example 3. This run takes the same time, as a computation without that option would take.

Remarks

During the work with *ReaSyn* it turned out, that the colorspace reduction did not have the desired effect, as the problems that are solvable in reasonable time

do not include many colors.

Best results were obtained, when *Deadend* optimization was called before *EnvWin* optimization. This is justified by the fact, that building the dead end attractor is much faster than solving the whole game. After the nodes found by *Deadend* optimization are removed, the remaining game is smaller and can be solved much faster. Both optimizations can be switched on or off by commandline options. Please refer to the user interface description by Jens Regenber [Reg05].

For the running time of the used algorithms it should be mentioned, that the formula game generation as done by [Reg05] is twice exponential if a determinization is needed. This can be circumvented by directly giving a deterministic Büchi automaton as specification. Division is done in linear time but glue is n -exponential for n glueing operations, as it is a determinization algorithm.

Chapter 8

Summary

A reactive systems was introduced as an architecture, that defines a graph with processes as vertices and signals as edges. A specification on the signals issues how the reactive system should behave. An implementation for each processor such that the overall behaviour respects the specification is a solution to the distributed synthesis problem.

In this thesis an approach to find such an implementation for each processor is presented. To achieve this, the distributed system is first modelled as a distributed game with n players. After simplifications of the game the number of players is successively reduced until a two-player game is obtained, the two-player game is algorithmically solved.

This procedure provides a solution to the distributed synthesis problem. As *ReaSyn* and all used implementations are provided as a framework to model distributed systems by architecture and specification definitions, this also shows the practical feasibility of the distributed synthesis problem. All datastructures and algorithms are implemented in an efficient way. To encourage further development the classes are provided as well documented source code.

As no theoretical background is needed to work with *ReaSyn* it can be handled by unexperienced personell for realizability tests and prototyping.

ReaSyn was implemented in close collaboration with Jens Regenber [Reg05]. Recall the three parts as depicted in Figure 1.1. The functions and simplifications were adapted several times to gain the best overall performance.

Two clearly defined interfaces between the layers as depicted in Figure 1.1 made a fast development possible. As needs of the later layers demanded the before layers to propagate specific information, some of the specifications had to be adapted several times during the development phase. For example the code generation needs information about the values of the environment variables to be encoded in the distributed game nodes. Another example for the

close connection of the parts is, that changes in the construction of the specification games had immense influence on the size of the distributed games.

This thesis was a large practical project. Almost 20.000 lines of program code were written and the compiled program has a size of 4 megabytes. As it was not sure at the beginning, if the resulting program would be of any practical use the thesis was quite a challenge. It should also be mentioned, that the several optimizations in the parts of the overall programs caused interdependencies that sometimes minimized the results of former optimizations. Also the arrangements during the development phase were demanding.

The high theoretical complexity of the distributed synthesis problem lead to refrainment from giving an implementation until now. Thus the results of this work are quite pioneering as it is the first implementation to show that with efficient optimizations and careful construction testing for realizability and even synthesis is possible.

With *ReaSyn* a tool is provided for everyone to test the results and possibilities of this thesis. It states, that the distributed synthesis problem is solvable in practice.

Future Work

Efficient construction and simplification of the distributed games, as it is mentioned in the before chapters, strongly reduces the size of the problem. However, a further simplification could be investigated to solve even more complex systems. A big problem in this context is, that eliminating to many nodes from the distributed game leads to wrong synthesis results.

Two reliable algorithms were given to reduce the number of nodes in a game. Further test could investigate the effects of the *EnvWin* simplification inbetween glueing steps. This may not always be effective, as if unsuccessful the game was solved without benefit. A possibility to predict the account of the optimization would be very useful.

An often used approach to simplify games is splitting the game into strongly connected components (SCC) and do simplifications therein. SCC and other graph algorithmic approaches could be investigated.

Color reduction could be used on SCCs to further minimize the occurring colors.

The generated strategies could be enlarged by edges, that start from a node that is winning for the player. Note that the edges can only be added, if they do not create a cycle.

Appendix A

Classes

To implement the above mentioned definition of distributed games, an efficient datastructure was developed. The following list shows the classes, that are directly connected to the tasks of this thesis. *ReaSyn* also contains classes for different automata and code generations. They are used and developed by [Reg05] and therefore not included here.

Figure A.1 shows the class interaction as UML-diagram.

A.1 Class dependabilities

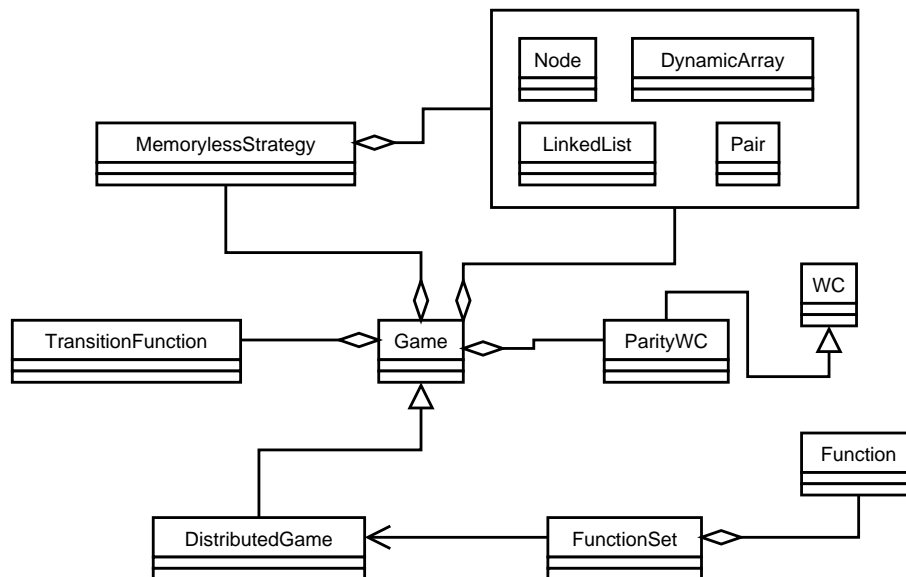


FIGURE A.1 UML-Overview

In the upper right corner of the figure are the basic datastructures, that were used in multiple classes.

A.2 Basic datastructures

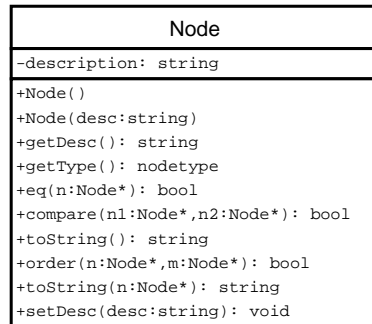


FIGURE A.2 Class Node

The class *Node* (figure A.2) is used to represent the game nodes. It contains a description for identification and a typedef that defines the type of the node. A *Node* can be of one of the following types: NODE, PAIRNODE, SETNODE, DISTNODE and PARITYNODE.

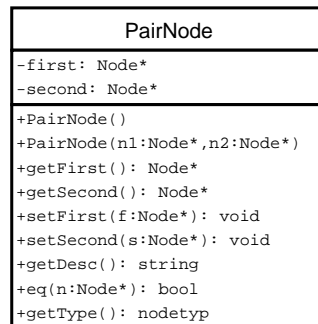


FIGURE A.3 Class PairNode

PairNodes (figure A.3) combine classes *Pair* and *Node* to deliver a pair of nodes, where the contents can be accessed by the provided member functions.

A *SetNode* (figure A.4) contains a list of pointers to nodes. They are internally referred to as a *LinkedList*. Adding a node to the set does not check if the nodes is already contained. The *SetNode* is sorted according to the node description.

Distributed game nodes can be represented by *DistNode* (cf. figure A.5), that inherits from class *Node*. A *DistNode* contains multiple nodes and an attribute *size* telling how many nodes are included. There are two possible access op-

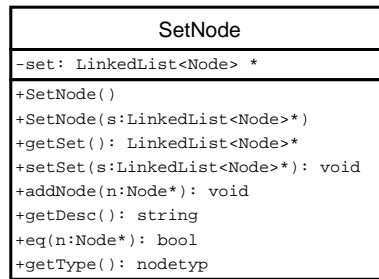


FIGURE A.4 Class SetNode

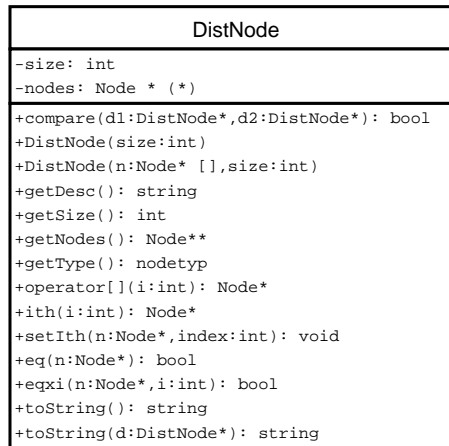


FIGURE A.5 Class DistNode

erations provided: *operator[]* (int *i*) and *ith*(int *i*), where *operator[]* (int *i*) can be used as with conventional arrays. “[*i*]” evoked on a *DistNode* will return the same as the usual “->ith(*i*)”.

A.2.1 Template datastructures

The following datastructures can contain elements of any type. In the UML-diagrams the type is noted with the placeholder T, indicating a template. By instantiating an object of the following datastructures, the type is given. For example

```
DynamicArray<int> *array = new DynamicArray<int>(2);
```

creates a new *DynamicArray* for integer values of size 2.

Pair A.6 is mainly used to represent pair nodes in the glue procedure.

For list representation two datastructures were implemented: *LinkedList* (figure A.7) and *DynamicArray* (figure A.8). Both classes provide a sorting al-

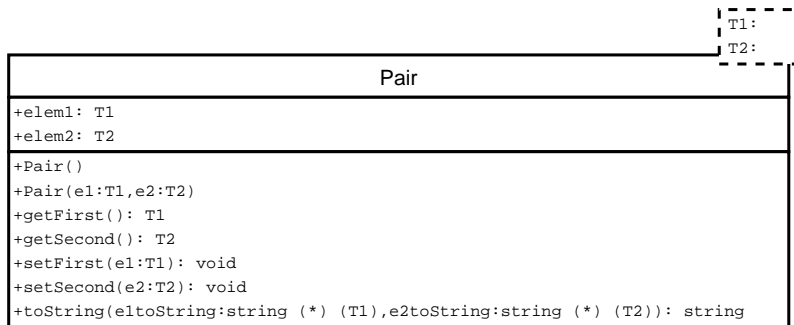


FIGURE A.6 Class Pair

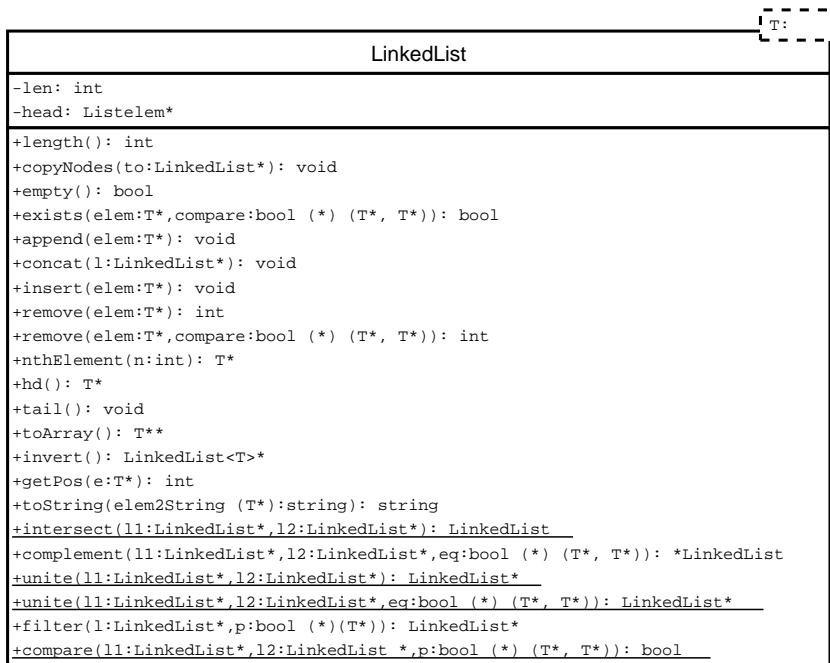


FIGURE A.7 Class LinkedList

gorithm, that given a equivalence relation sorts the datastructure. The size of a *LinkedList* is flexible and the length updated in a member variable. Additionally two pointers link to the head and tail of the list. With the function *toDynamicArray* a *LinkedList* can be transformed into a *DynamicArray*.

The size of a *DynamicArray* is doubled each time an element is added with no space left in the *DynamicArray*. Also, if the size occupied is lower than half of the available space, the array's size is halved. This is always checked when an element is deleted from the *DynamicArray*. It therefore has two member variables keeping track on how many elements are included and how many space is reserved. Provided a relational and a comparison function, *DynamicArrays* provide binary search. The access to an element is thus much faster as for

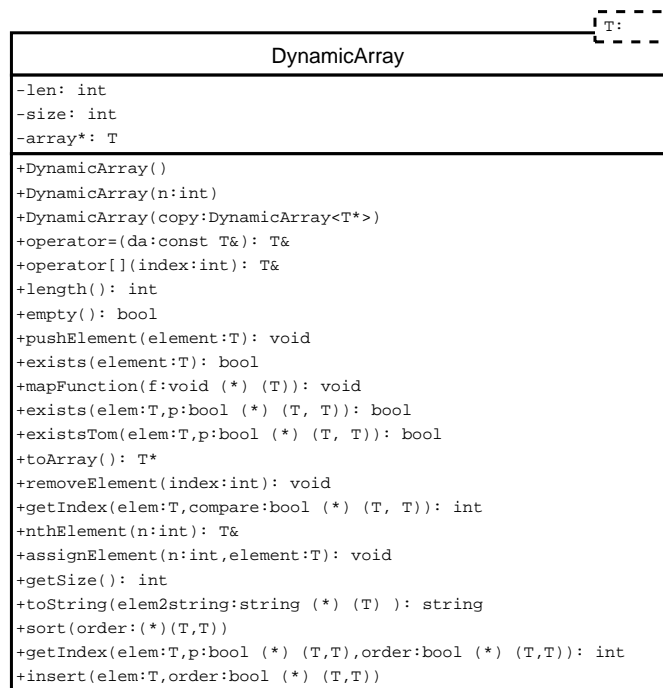


FIGURE A.8 Class DynamicArray

LinkedList. Note, that *DynamicArrays* can also be used without binary search, if the speedup is not required or no relational and comparison functions are applicable.

A.3 Advanced datastructures

Advanced datastructures provide the possibility to model transition functions as well as functions of the processes. *FunctionSet* represents a set of functions.

TransitionFunction
<pre>#size: int #nodes: DynamicArray<Node*> #matrix: DynamicArray<LinkedList<Node> *></pre>
<pre>+getSize(): int +setEdge(from:Node *,to:Node *): void +deleteEdge(from:Node *,to:Node *): void +isEdge(from:Node *,to:Node *): bool +addNode(n:Node *): void +removeNode(n:Node *): void +printMatrix(wd:bool=true): void +toString(wd:bool=true): string +getSuccessors(n:Node *): LinkedList<Node> * +getPredecessors(n:Node *): LinkedList<Node> * +getNodes(): LinkedList<Node> *</pre>

FIGURE A.9 Class TransitionFunction

Function
<pre>+domain: DynamicArray<string*> +image: DynamicArray<string*> +images: DynamicArray<string*></pre>
<pre>+Function(domain:DynamicArray<string*>,image:DynamicArray<string*>) +Function(domain:DynamicArray<string*>,image:DynamicArray<string*>,images:DynamicArray<string*>) +getDomain(): DynamicArray<string*> +f(x:string): string +def(x:string,y:string): void +setImages(images:DynamicArray<string*>): void +toString(): string +toString(f:Function*): string</pre>

FIGURE A.10 Class Function

FunctionSet
<pre>-fset: FunctionSet* -fns: DynamicArray<Function *></pre>
<pre>-FunctionSet() +getInstance(): FunctionSet* +addFunctions(input:DynamicArray<string*>,output:DynamicArray<string*>): DynamicArray<string*> +getFunction(fn:string): Function* +getAllFunctions(): DynamicArray<string*> +toString(): string +findTotalFunction(nodes:DynamicArray<Node*>): void +functionSimulator(input:DynamicArray<string*>,output:DynamicArray<string*>): string +findFunction(n1:Node*,n2:Node*): string +findAllFunctions(n1:Node*,n2:Node*): DynamicArray<string*> +simulator(in:DynamicArray<string*>,out:DynamicArray<string*>,fun:string): bool +genNewStartVals(x:DynamicArray<string*>,fun:string): string*</pre>

FIGURE A.11 Class FunctionSet

A.4 Game datastructures

The class *Game* A.12 represents lists of start, player and environment nodes, a transition function of class *Transitions* (cf. figure A.9) and a parity winning condition of class *WC* (cf. figure A.14). One helpful member function is *toGDL*. It produces a game graph representation in **.gdl** format as used by the graph tool **aiSee** [aAIG]. It is also used, when the option `-gdl` is given to *ReaSyn* by the commandline.

Game
<pre> -start: LinkedList<Node> * -player: LinkedList<Node> * -environment: LinkedList<Node> * -functions: TransitionFunction * -w: ParityWC * +addPlayerNode(n:Node *): void +addEnvironmentNode(n:Node *): void +addTransition(from:Node *,to:Node *): void +addStartNode(s:Node *): void +setParity(n:Node *,parity:int): void +getParity(n:Node *): int +getMaxParity(): int +getMinParity(): int +toString(verbose:bool=true): string +removePlayerNode(n:Node *): void +removeEnvironmentNode(n:Node *): void +removeStartNode(n:Node *): void +removeTransition(from:Node *,to:Node *): void +getPlayerNodes(): LinkedList<Node> * +getEnvironmentNodes(): LinkedList<Node> * +getTransitionFunction(): TransitionFunction * +getStartNodes(): LinkedList<Node> * +getWinningCondition(): ParityWC * +setTransitionFunction(t:TransitionFunction *): void +setPlayerNodes(p:LinkedList<Node> *): void +setEnvironmentNodes(e:LinkedList<Node> *): void +isPlayerNode(n:Node *): bool +isEnvironmentNode(n:Node *): bool +isStartNode(n:Node *): bool +isGameNode(n:Node *): bool +subgame(u:LinkedList<Node> *): Game * +solve(w[2]:LinkedList<Node> *,w[2]:MemorylessStrategy *,t:type=Game::MAXPARITY): void +removeDeadends() +removeEnvWin() </pre>

FIGURE A.12 Class Game

WC is kept generic, as within *ReaSyn* there are various winning conditions due to the automaton representation of the specification.

The class *ParityWC* keeps track on the minimum and maximum occurring colors and assigns them to nodes.

Distributed games are represented by the class *DistributedGame* (cf. figure A.13), which inherits from *Game* and gains some extra information about the number of local games, the distributed game itself and information on how the

number of games can be transformed.

Edges between game nodes are internally maintained through the class *TransitionsFunction*. The edges are stored in matrix representation. This allows to look up an edge between two arbitrary nodes can be very fast.

Functions, which map input strings on output strings are provided by class *Functions*. They are used to map a processes' input signals to output signals.

The collection of all available functions is stored in *FunctionSet* (cf. figure A.11).

The memoryless strategy is represented by class *MemorylessStrategy* (cf. figure A.16). It has a list of winning successors for a node.

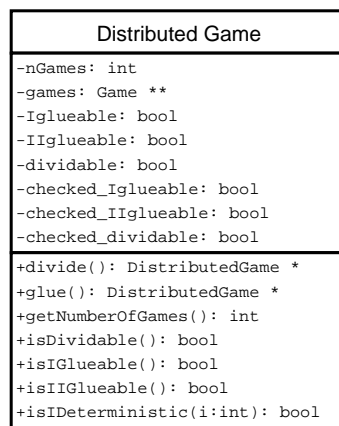


FIGURE A.13 Class DistributedGame

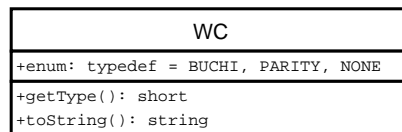


FIGURE A.14 Class WC

ParityWC
<code>-nodes: DynamicArray<Node *></code> <code>-colors: DynamicArray<int></code> <code>+maxpar: int</code> <code>+minpar: int</code>
<code>-getIndex(n:Node*): int</code> <code>+ParityWC()</code> <code>+ParityWC(l:LinkedList<Node*>)</code> <code>+getType(): short</code> <code>+getMaxParity(): int</code> <code>+getMinParity(): int</code> <code>+addNode(n:Node*,color:int=-1, safe:bool=true): void</code> <code>+removeNode(n:Node*): void</code> <code>+setColor(n:Node*,color:int): void</code> <code>+getColor(n:Node*): int</code> <code>+isAccepted(infinitySet:LinkedList<Node*>): bool</code> <code>+toString(): string</code>

FIGURE A.15 Class ParityWC

MemorylessStrategy
<code>-nodes: DynamicArray<Node *></code> <code>-succ: DynamicArray<LinkedList<Node> *></code>
<code>+MemorylessStrategy()</code> <code>+addSuccessor(node:Node*,succ:Node*): void</code> <code>+addSuccessor(node:Node*,succs:LinkedList<Node*>): void</code> <code>+removeSuccessor(node:Node*,nsucc:Node*): void</code> <code>+getSuccessor(node:Node*): LinkedList<Node>*</code> <code>+getNumberOfSuccessors(node:Node*): int</code> <code>+toString(): void</code>

FIGURE A.16 Class MemorylessStrategy

List of Figures

1.1	Interaction between the two theses	3
2.1	The partition of <i>ReaSyn</i>	7
3.1	An example parity game	13
3.2	An example two-player game	16
3.3	Construction of $X_{1-\sigma}$ and X_σ [MW03]	16
3.4	Winning regions algorithm applied on Figure 3.2	17
3.5	<code>winning_regions</code>	19
3.6	<code>win_opponent</code>	20
4.1	Two local games	24
4.2	Distributed Game \mathcal{G} constructed from local games $\mathcal{G}_0, \mathcal{G}_1$	24
4.3	Distributed Game \mathcal{G} constructed from local games $\mathcal{G}_0, \mathcal{G}_1$	25
4.4	Distributed game and divided distributed game	27
4.5	Two local games	29
4.6	The Distributed Game	29
4.7	Glued Distributed Game	29
5.1	n -process pipeline architecture	34
5.2	One-process Architecture	35
5.3	A process game	35
5.4	A simple specification game	37
6.1	Enriching a formula game	43
6.2	A distributed game after construction	46
6.3	Result of dead end removal	47
6.4	DivideNode pseudo code	48
6.5	divide pseudo code	49
6.6	glue pseudo algorithm	49
6.7	<i>pSucc</i> pseudo algorithm	52
6.8	<i>eSucc</i> pseudo code	53
7.1	2-process pipeline	54
7.2	PROMELA example	56
7.3	3-process pipeline	57
7.4	3-process pipeline	58

A.1	UML-Overview	62
A.2	Class Node	63
A.3	Class PairNode	63
A.4	Class SetNode	64
A.5	Class DistNode	64
A.6	Class Pair	65
A.7	Class LinkedList	65
A.8	Class DynamicArray	66
A.9	Class TransitionFunction	67
A.10	Class Function	67
A.11	Class FunctionSet	67
A.12	Class Game	68
A.13	Class DistributedGame	69
A.14	Class WC	69
A.15	Class ParityWC	70
A.16	Class MemorylessStrategy	70

List of Tables

7.1	Results of $\Box(x = y)$	55
7.2	Results of $\Box(x \rightarrow \Diamond y)$	55
7.3	Results of $\Box(y)$	57
7.4	Results of $\Box(y) \wedge \Diamond(y = 0)$	58

Bibliography

- [aAIG] absInt Angewandte Informatik GmbH. aisee homepage.
<http://www.aiSee.com>.
- [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.
- [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.
- [MAW89] L. Lamport M. Abadi and P. Wolper, editors. *Realizable and unrealizable specifications of reactive systems*. Lecture Notes in Computer Science. ICALP'98, 1989.
- [MR] Tobias Maurer and Jens Regenber. Reasyn homepage.
<http://www.ReaSyn.org>.
- [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. Multiple person alternation. In *20th IEEE Symposium Foundations of Computer Science*, pages 348–363. FOCS, 1979.
- [PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *ACM POPL*, pages 179–190. ACM, 1989.

- [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.
- [Reg05] Jens Regenberg. Synthesis of reactive systems. Master's thesis, Universit"at des Saarlandes, regenberg@react.cs.uni-sb.de, 2005.
- [Saf88] S. Safra. On the complexity of ω -automata. In *IEEE Symposium on the Foundations of Computer Science*, pages 319–327, 1988.
- [Tho95] W. Thomas, editor. *On the synthesis of strategies in infinite games*. Lecture Notes in Computer Science. STACS'98, 1995.
- [Zie98] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science 200*, pages 135–183, 1998.