Improving the Efficiency and Quality of Omega-Regular Synthesis

Saqib Sohail
University of Colorado Boulder, saqib.sohail@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/ecen_gradetds
Part of the Computational Engineering Commons, and the Computer Engineering Commons

Recommended Citation
Sohail, Saqib, "Improving the Efficiency and Quality of Omega-Regular Synthesis" (2014). Electrical, Computer & Energy Engineering Graduate Theses & Dissertations. 84.
https://scholar.colorado.edu/ecen_gradetds/84

This Dissertation is brought to you for free and open access by Electrical, Computer & Energy Engineering at CU Scholar. It has been accepted for inclusion in Electrical, Computer & Energy Engineering Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Improving the Efficiency and Quality of Omega-Regular Synthesis

by

Saqib bin Sohail

B.S. Elect. Engineering, University of Engineering & Technology, Lahore 2002

M.S. Elect. Engineering, University of Colorado at Boulder, 2006

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2014
This thesis entitled:
Improving the Efficiency and Quality of Omega-Regular Synthesis
written by Saqib bin Sohail
has been approved for the Department of Electrical, Computer, and Energy Engineering

__________________________________________

Fabio Somenzi

__________________________________________

Pavol Cerny

Date _______________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
Sohail, Saqib bin (Ph.D., Electrical and Computer Engineering)

Improving the Efficiency and Quality of Omega-Regular Synthesis

Thesis directed by Professor Fabio Somenzi

The automatic synthesis of a program from its specification has been an ambitious goal since Alonzo Church asked in 1962 if given a Monadic Second Order (MSO) formula as a logical specification of reactive system with inputs $I$ and outputs $O$ then is it possible to decide where there exists a reactive system that satisfies the specification; and if the specification is realizable then a reactive system should be generated as a proof. Büchi and Landwaber proved in 1969 that solving the automatic synthesis problem is decidable. While in 1989 Pnueli and Rosner described an automatic synthesis procedure with the best complexity bounds to date.

Since then there have been significant advancements in developing methodologies for the automatic synthesis of an arbitrary MSO formula as well as special cases which are practically relevant. The current focus has been on improving the efficiency of the automatic synthesis procedures. However, all the current techniques generate an implementation as a proof which is significantly larger than a handwritten implementation. Thus making the automatic synthesis of reactive systems from a logical specification unattractive for industrial adoption. (In addition, writing a complete logical specification of a reactive system is not a trivial task.)

The focus of this thesis is to improve the quality of the implementation generated by an automatic synthesis approach as a proof of realizability of the specification. In all the automatic synthesis approaches, the logical specification is converted to a two player game (the size and complexity of the game is different) where the objective of the antagonist is to find a strategy through which the specification cannot be satisfied while the objective of the protagonist is to find a strategy through which the satisfaction of the specification is guaranteed. In this thesis it is argued that the conversion of the specification to a game is crucial step for both the efficiency of the synthesis approach and quality of the implementation generated as a result.

After a thorough investigation into the reasons why the current approaches fail at generating a reason-
ably sized implementation, a framework has been proposed through which the specification is converted to a significantly more compact game that drastically improves the efficiency and the quality of the implementation generated as a proof. Furthermore, this framework does not compromise the efficiency of the automatic synthesis approach when the specification is restricted to a GR(1) specification.

Experiments show the main advantage of the automatic synthesis approach based on this framework. Furthermore, many of the salient features of this framework can be adopted by other automatic synthesis approaches.
Acknowledgements

In the name of Allah, the most Gracious and the most Compassionate. I am extremely grateful to Him for surrounding me with wonderful people who have helped and supported me in successfully completing this thesis.

First of all, I would like to thank my adviser Professor Fabio Somenzi for his guidance, support, and wisdom throughout my endeavors. Despite all my flaws, he never for a moment withdrew his support and was always readily available. I would like to thank my parents who fully supported me in my decision to pursue an advanced degree and never inquired about completion of my degree. I would like to extend my gratitude to Professor Shahid H. Bokhari who inspired me to develop a deeper understanding of computer engineering and that led me to discover my current interests. Finally, I would like to give a shout-out to my wife and my kids who at times were not happy with my long hours but were always there with me on this long ride.
# Contents

## Chapter

1. **Introduction** 1  
   1.1 Background .................................................. 3  
   1.2 Thesis Contributions ........................................... 5  
   1.3 Thesis Organization ............................................. 8  

2. **Automata, Logic and Games** 10  
   2.1 Finite Transition Structures, Automata and Transducers ................. 10  
      2.1.1 Transducers and Hardware ....................................... 12  
   2.2 Alternating and Universal Automata .................................. 13  
   2.3 Omega Automata .................................................. 16  
   2.4 Nongeneralized Parity Automata ..................................... 20  
   2.5 LTL: Linear Time Logic ........................................... 24  
   2.6 Borel Hierarchy and Omega-Regular Languages ......................... 27  
      2.6.1 Syntactic Characterization of LTL ............................... 28  
      2.6.2 Omega Automata and Borel Hierarchy ........................... 29  
   2.7 Two-Player Games ................................................. 30  
      2.7.1 Specifications and Moore Machines ............................ 35  
   2.8 Symbolic Representations and Variable Ordering ....................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Synthesis of Reactive Systems</td>
<td>43</td>
</tr>
<tr>
<td>3.1</td>
<td>Satisfiability vs. Realizability</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Game Based Approaches</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Synthesis from General Reactive (1) Specifications</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>LTL to Automaton</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>Determinization and Games</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>To Determine or Not to Determine?</td>
<td>54</td>
</tr>
<tr>
<td>3.7</td>
<td>Determinization of an NBW</td>
<td>57</td>
</tr>
<tr>
<td>3.8</td>
<td>Specialized Determinization Procedure</td>
<td>61</td>
</tr>
<tr>
<td>3.9</td>
<td>Solving Parity Games</td>
<td>63</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Nongeneralized Parity Games</td>
<td>63</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Conjunctive and Disjunctive Parity Games</td>
<td>69</td>
</tr>
<tr>
<td>3.10</td>
<td>Safraless Approaches</td>
<td>75</td>
</tr>
<tr>
<td>3.11</td>
<td>Efficiency of Symbolic Algorithms</td>
<td>79</td>
</tr>
<tr>
<td>4</td>
<td>The Hierarchical Approach</td>
<td>81</td>
</tr>
<tr>
<td>4.1</td>
<td>Classification of Deterministic Omega Games</td>
<td>82</td>
</tr>
<tr>
<td>4.2</td>
<td>Games with Conjunctive or Disjunctive Parity Conditions</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>Assumptions and Guarantees</td>
<td>91</td>
</tr>
<tr>
<td>4.3.1</td>
<td>GR(1) Specifications</td>
<td>92</td>
</tr>
<tr>
<td>4.3.2</td>
<td>General Case</td>
<td>101</td>
</tr>
<tr>
<td>4.4</td>
<td>General Reactive(n) Specification</td>
<td>103</td>
</tr>
<tr>
<td>4.5</td>
<td>Well-Separated Environment</td>
<td>107</td>
</tr>
<tr>
<td>5</td>
<td>Transition Constraints</td>
<td>109</td>
</tr>
<tr>
<td>5.1</td>
<td>Challenges in the Encoding of Transition Structures</td>
<td>111</td>
</tr>
<tr>
<td>5.1.1</td>
<td>State Encodings of Automata and Games</td>
<td>114</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Bisimilar States in Automata and Games</td>
<td>117</td>
</tr>
</tbody>
</table>
5.1.3 Deterministic Automata with Pseudo Inputs ............................................. 120
5.2 $R$-Generability and 1-Definiteness .......................................................... 122
5.3 LTL and $R$-Generability ......................................................................... 131
5.4 General Safety Properties ....................................................................... 133
5.5 Realizability of Relation-Based game $\hat{G}_\varphi^R$ .................................. 140
5.6 Solving the Relation-Based game $\hat{G}_\varphi^R$ .......................................... 144
5.7 Synthesis from Transition Constraints ...................................................... 147
5.8 Boolean Equations .................................................................................... 149
5.9 Correctness and Importance of Parameterized Solutions ......................... 155
5.10 Obligation Properties and Transition Constraints .................................... 160
5.11 Relation-Based approach and common Specifications .............................. 164
5.12 Optimizing the Parametric Representation ............................................. 166
  5.12.1 Flow Networks and Binary Decision Diagrams ................................. 171
  5.12.2 Factorization and Flow Network ......................................................... 171
  5.12.3 Retiming and Automata ................................................................. 173

6 Input-Based Games .................................................................................... 177
  6.1 Input-Based Games vs. Turn-Based Games ............................................. 177
  6.2 Solving Input-Based Games ................................................................... 187
  6.3 Mealy Coloring Functions .................................................................... 199

7 Specifications that are not General Reactivity (1) .................................. 201
  7.1 Simplified AMBA: Shared-Bus Architecture ......................................... 202
  7.2 Simplified AMBA: Complete Crossbar Architecture ............................ 206
  7.3 Simplified AMBA: Partial Crossbar Architecture .................................. 208
  7.4 Reduction of General Reactive (2) to General Reactive (1) ................. 212
# Experimental Results

8.1 Importance of Retiming ......................................................... 217

8.2 Optimization through ABC .................................................... 219

# Conclusions and Outlook

9.1 Thesis Conclusions .............................................................. 225

9.2 Future Work ................................................................. 226

# Bibliography


### Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The description various symbols used in this chapter.</td>
</tr>
<tr>
<td>5.2</td>
<td>When the environment chooses to play the prefix $\neg g, g, \neg g, \neg g, g$ of then the possible runs of $G_\varphi$ and the corresponding runs of $M_\varphi$ are shown here. The label $S_i$ denotes the state of $M_\varphi$ in the $i^{th}$ turn, while $P_i$ denotes the value of the parameters in the $i^{th}$ turn, and $X_{si}$ denotes the value of the output variable in the $i^{th}$ turn.</td>
</tr>
<tr>
<td>7.1</td>
<td>Shared-bus architecture between four masters and a single slave.</td>
</tr>
<tr>
<td>7.2</td>
<td>Crossbar architecture between four masters and a single slave with two identical ports.</td>
</tr>
<tr>
<td>7.3</td>
<td>Crossbar architecture between four masters and a single slave with two identical ports. $\text{MASTER}_0$ and $\text{MASTER}_3$ can only read through Port-0 of the memory while $\text{MASTER}_1$ can only read through Port-1 of the memory. If $\text{MASTER}_2$ is requesting a locked access then it will only be given such an access through Port-1.</td>
</tr>
<tr>
<td>8.1</td>
<td>Partial LTL Specification of $n$-Client AMBA Bus Arbiter</td>
</tr>
<tr>
<td>8.2</td>
<td>Experimental Results AMBA Specification [BGI07a]</td>
</tr>
<tr>
<td>8.3</td>
<td>Experimental Results AMBA Specification [GCH11]</td>
</tr>
<tr>
<td>8.4</td>
<td>Experimental Results GENERAL BUFFER Specification [BGI07b]</td>
</tr>
<tr>
<td>8.5</td>
<td>Comparison of Results from ANZU for AMBA Specification [BGI07a]</td>
</tr>
</tbody>
</table>
8.6 ANZU_R represents the results of the ANZU tool when the \( \mathcal{R} \)-generable properties of the specification (which are not transition constraints) are replaced manually with their respective transition constraints. .......................................................... 223

8.7 Comparison of Results from ANZU for AMBA Specification [GCH11] .................................. 223

8.8 ANZU_R represents the results of the ANZU tool when the \( \mathcal{R} \)-generable properties of the specification (which are not transition constraints) are replaced manually with their respective transition constraints. .......................................................... 223

8.9 Comparison of Handwritten results of AMBA Specification [BGJ+07a] with before and after ABC Optimizations .......................................................... 224

8.10 The AFTER results have been computed by running the resyn2 command followed by the fpga command in ABC. .......................................................... 224
Figures

Figure

2.1 An automaton with a conjunctive parity condition. ........................................... 23
2.2 A non-generalized parity automaton which is language equivalent to the conjunctive parity automaton of Figure 2.1. The underscore indicates the hit position. As there were no states that were assigned priority 2 or 3, the priorities 4 and 5 got bumped down to 2 and 3 respectively. ................................................................. 24
2.3 Automaton that accepts $\{\neg a, a\}^\omega$. ......................................................... 26
2.4 First few levels of Borel hierarchy. ................................................................. 28
2.5 (a) An input-based game obtained from $\varphi$. (b) A turn-based game obtained from $\varphi$. ........ 41
2.6 $\mathcal{G}((h \rightarrow g_0 \rightarrow X \neg m) \land (h \rightarrow g_1 \rightarrow X m) \land ((g_0 \land g_1) \leftrightarrow X p))$. .............. 42
2.7 The BDDs for $\Gamma_2(S)$, $\Gamma_1(S)$ and $\Gamma(S)$ with variable order $x_1, x_0$. ............ 42
2.8 The BDDs for the function $f$ with two different variable orders, $f = (a \land b) \lor (a \land c)$ . . . 42

3.1 Automaton for $\mathcal{G}(p \rightarrow X F q)$ with acceptance conditions on the transitions (left) and with conditions moved to the states (right). ................................................................. 49
3.2 The language accepted by both Büchi automata is $bd^\omega \lor bd^\omega$. ............................. 52
3.3 Every run of $G$ is an accepting run. The nondeterministic Büchi automaton $A_\varphi$ accepts the language defined by the LTL formula $F G p \lor F G q$. The nondeterministic game $G$ is the composition of $G$ and $A_\varphi$. ................................................................. 53
3.4 Every run of $G$ is an accepting run. The deterministic co-Büchi automaton $A_\varphi$ accepts the language defined by the LTL formula $F G p \lor F G q$. The initial state of $A_\varphi$ is the non-accepting state. The deterministic game $G$ is the composition of $G$ with $A_\varphi$. .......................... 54

3.5 Both $A^d_\varphi$ and $A^n_\varphi$ are co-Büchi automata and $A^d_\varphi \preceq_f A^n_\varphi$. .......................... 55

3.6 The NBW $A$ accepts the language $\Sigma^* p^\omega$ and the automaton $D$ is generated by the subset construction procedure applied to $A$. .......................... 58

3.7 Zielonka’s algorithm for nongeneralized parity games. $d(G)$ is the largest priority of $G$. $A_d(G)$ is the set of states of $G$ assigned the priority $d$. .......................... 64

3.8 A parity game. Priority of states are shown in parentheses .......................... 66

3.9 A parity game with four colors: Priority of states are shown in parentheses. The circle player wins. .......................... 68

3.10 Chatterjee’s algorithm for conjunctive and disjunctive parity games. $A(G)$ is the set of states of game $G$. $d(G, \pi)$ is the highest priority of $G$ with respect to $\pi$. $A_d(G, \pi)$ is the set of states assigned the priority $d$ by $\pi$ in the game $G$. .......................... 70

3.11 A simple generalized parity game with the winning condition defined by $G F b \lor G F c$. .......................... 71

3.12 A conjunctive parity game with both conditions have four colors: Priority of states are shown in parentheses. The disjunctive player which controls the square states wins. .......................... 72

3.13 A conjunctive parity game with both conditions have four colors: Priority of states are shown in parentheses. The conjunctive player which controls the circle states wins. .......................... 73

3.14 $\varphi = G F p$. The NBW for $\neg \varphi$ and UCW for $\varphi$. The label of each state reflects the priority of the state. The transitions that are connected (for instance $1 \rightarrow 1 \land 1 \rightarrow 2$) need to be simultaneously taken; this is the reason why the run of a universal automaton is a tree. .......................... 78

3.15 UKCW with $K = 2$ is a safety automaton and $(0, 3)$ is the trap state. The state with the label $\infty$ represent the initial state. .......................... 78

4.1 Equivalent Games .......................... 86
4.2 $G_\varphi$ where $\varphi = G(f \to F w), G_\psi$ where $\psi = G F(w \lor l), G_{\varphi \land \psi} = G_\varphi \times G_\psi$ with conjunctive parity condition, $G'_{\varphi \land \psi} = G_\varphi \times G_\psi$ with non-generalized parity condition. 89

4.3 An incremental algorithm for Conjunctive Parity Games. $S : M \times (U_1 \cap Q_1) \to (U_1 \cap Q_0)$ is a strategy, where $M$ is the finite memory. 90

4.4 Iterative algorithm to remove protagonist’s easy-to-identify losing states. The procedure `createstrategy` is an implementation of (4.5). 91

4.5 Various games corresponding the assumptions in $S_e \cup L_e$. 96

4.6 An incremental algorithm for Conjunctive Parity Games. $S : M \times (U_1 \cap Q_1) \to (U_1 \cap Q_0)$ is a strategy, where $M$ is the finite memory. 98

4.7 (a) The game $G_{ss}$ and the game $G_e$ are composed with each other; after playing the composed game, one finds the winning strategies for the specification $S_e \land L_e \rightarrow S_s$ when the environment does not violate the antecedent ($S_e \land L_e$). (b) The game $G_s$ which is obtained from $G_e \times G_e$ (the conjunctive parity condition of $G_e$ has been negated) by removing the system’s losing states (the unreachable states are marked with yellow color). (c) Notice that the system cannot win $G_s$; however it can win $G_e \times G_s$. (d) In later examples, the unreachable states of $G_s$ will not be shown and the last component of the generalized parity condition is redundant. The coloring function in this game is an example of a generalized parity condition where the inner parentheses imply a conjunction while the outer parentheses imply a disjunction. 99

4.8 In the generalized parity game $G = G_s \times G_{L_1} \times G_{L_2} \times G_{L_3}$ every state in $G$ is connected to every other state (the unreachable states of $G$ are not shown). The states have been marked with the label of the incoming transitions (in this particular game every incoming transition of a state has the same label, which is simply a coincidence). The generalized priority condition of every state is shown under the state. 100

4.9 The counter to convert the parity conditions corresponding $L_s$ to a non-generalized parity condition. 101
4.10 The game $G'$ obtained by composing the counter in Figure 4.9 with $G$ (the unreachable states of $G'$ are not shown). Instead of having the transitions drawn, these have been listed. For instance the state $A_0$ implies that the token is in state $A$ of $G$ and $0^{th}$ state of the counter; this state has transitions to states $A_0$, $B_1$, $C_1$, $D_0$, and $E_0$. The label of the transition from $A_0 \rightarrow B_1$ is $B$ (which represents $\neg a \land b \land \neg x$ as listed in Figure 4.8).

4.11 The conjunctive parity game $\tilde{G}$ is obtained from $G'$ shown in Figure 4.10 by invoking Lemma 4.3.2.

5.1 Büchi Automaton for $G F p$.

5.2 (a) $\Phi_1 = G((a \land b) \rightarrow X(c \land \neg a))$ (b) $\Phi_2 = G((a \land \neg c) \rightarrow (b \leftrightarrow X b))$ (c) $\Phi = \Phi_1 \land \Phi_2$. There are two unreachable states in $A_\Phi = A_{\Phi_1} \times A_{\Phi_2}$ denoted by the set $U$, while the set $X$ denotes the don’t care states. When $A_\Phi$ is obtained from $\Phi$ directly then the encoding is shown as the labels of the states.

5.3 There are two unreachable states in $A_{\Phi} = A_{\Phi_1} \times A_{\Phi_2}$ denoted by the set $U$, while the set $X$ denotes the don’t care states. $A_\Phi$ can be re-encoded with using two state variables as well.

5.4 (a) $\Phi_1 = G((\neg a \land c) \rightarrow ((a \lor \neg c) \land b))$ (b) $\Phi_2 = G(\neg a \rightarrow X b)$ (c) $\Phi = \Phi_1 \land \Phi_2$. There are two unreachable states in $A_\Phi$ which are not shown.

5.5 Irredundant automaton obtained from $A_\Phi$ of Figure 5.4.

5.6 $\varphi_1 = G((\neg a \rightarrow (b \leftrightarrow X b)) \land (a \lor c))$.

5.7 Irredundant automata for three LTL formulae. All states have priority 1. The one on the left is 1-definite because even when the current state of the automaton is not known, reading the input a single letter is enough to determine the next state of the automaton. This automaton shows that both LTL formulae represent transition constraints (transition constraints are defined in Section 5.3). The automaton on the right is not definite because if the current state is not known then reading the input letter $(r \land \neg g)$ is not enough to compute the current state of this automaton.
5.8 (a) The incompatibility graph derived from the irredundant automaton for the LTL formula \( r W g \) shown in Figure 5.7. (b) The incompatibility graph derived from the irredundant automaton for the LTL formula \( G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \) shown in Figure 5.10.

5.9 Automata over \( \Sigma \) for \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \).

5.10 Automata over \( \hat{\Sigma} \) for \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \).

5.11 \( \varphi = \varphi_1 \land \varphi_2 \) where \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \).

5.12 The parameterized transducer \( M_{\varphi_1} \) that captures all the strategies to satisfy \( \varphi_1 \). The output function \( x \) is \( q \land x_i \).

5.13 The game \( G_\varphi \) obtained from the property \( \varphi = \varphi_1 \land \varphi_2 \), where \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \). In this game the environment moves after the system makes its move. The game \( G_\varphi \) is derived from \( G_\varphi \) by restricting it to system’s winning strategies.

5.14 The input-based game for \( G(r \lor g) \rightarrow G(r \land g \rightarrow Xg) \).

5.15 (a) The DAG of \( x' \) (b) The modified DAG of \( x' \).

5.16 (a) The BDD for \( x' = (a \land c) \lor b' \) with the variable order \( b', a, c \) (b) The BDD for \( x' = (a \land c) \lor b' \) with the variable order \( a, b', c \) (c) The unmodified DAG of \( x' \) derived from BDD in (b) (d) The modified DAG.

5.17 (a) The BDD for \( x' = (a \land c) \lor b' \) with the variable order \( b', a, c \) (b) The BDD for \( x' = (a \land c) \lor b' \) with the variable order \( a, b', c \) (c) The unmodified DAG of \( x' \) derived from BDD in (b) (d) The modified DAG.

6.1 The subgame of an input-based game induced by the removal of states in \( i \)-attraction depends on the value of \( i \).

6.2 The automata for \( G(a \rightarrow XF b) \). In automaton on the left, the color of the states are mentioned under the name of the states. In the automaton on the right, all the edges are colored 1 except the one labeled with a circle which is colored 0.
7.1 Shared-bus architecture between four masters and a single memory unit. Each master can also read the *hready* signal from the memory. That connection is not shown here. . . . . . 203
Chapter 1

Introduction

Today’s society heavily depends on technology and every aspect of life relies heavily on the ability of the electronic industry to innovate. A crucial step in the design and development of new systems is to verify whether a system exhibits the desired behavior in all conditions. The importance of this step can be gauged by the fact that it has become one of the central problems facing the electronic industry.

Currently, a correct implementation of a system is obtained after several design and verification cycles, where the implementation is refined until its verification does not fail. As the complexity of the systems increases, verifying the correct behavior of the implementations becomes even more important. Unfortunately, due to market demands the time from conception to production is getting shorter while the verification times are getting longer. Furthermore, the specification of the system is often incomplete and not every aspect of the implementation’s behavior can be verified.

Significant progress has been made in the formal verification of both finite and infinite state systems. The development of symbolic model checking, automatic abstraction refinement, abstract interpretation, and various satisfiability based techniques have helped the adoption of formal methods in industry, resulting in higher confidence that a system satisfies its specification.

One wonders if the current design and verification approach is always the most appropriate one. The automatic synthesis approach is to directly obtain a correct finite model from the specification; it generates a model using rules that guarantee that the model is an implementation of the desired system. Whenever the specification of a finite-state reactive system is given as a set of $\omega$-regular properties then the automatic synthesis approach is decidable. Unfortunately, automatically synthesizing a reactive system is significantly
harder than verifying an implementation against its specification. However when the system’s behavior is complex and the expected size of the implementation is small then one may opt for the automatic synthesis approach.

A reactive system is a type of system that continuously generates an output signal in response to an input signal. Controllers are special types of reactive systems, which are often used to provide access to shared resources like buses or memories. The implementation of a controller is often small yet significantly complex. A controller is often part of a larger system: its erratic behavior may result in performance degradation of the overall system or even lead to failure of the system. The main focus of this thesis is to develop an efficient automatic synthesis approach that can be used to synthesize reactive systems.

The first formal automatic synthesis framework based on the game-theoretic approach was proposed in [PR89]. The proposed approach accepts as an input a specification comprising a set of \( \omega \)-regular properties (namely LTL formulae). This implies that the specification describes a finite-state reactive system. If there exists a reactive system that satisfies the specification then this approach is able to find a finite-state implementation of that reactive system. The approach casts the problem of synthesizing a reactive system from its specification as a two-player game, where the antagonist controls the inputs while the protagonist controls the outputs. The game is played on a finite graph and the two players take turns in generating input and output signals respectively to produce a play, which is an infinite sequence of vertices of the graph. No matter how the antagonist generates the input signals, the protagonist must be able to generate the output signals such that the resulting play satisfies the specification. If the protagonist has a strategy to do so then it wins the game. If the protagonist wins the game then the specification is realizable; otherwise the antagonist wins the game, which implies that there does not exist a model that can satisfy the specification. An implementation can be obtained from one of the protagonist’s winning strategies. The game-theoretic approach to the synthesis of reactive systems is the focus of renewed attention thanks to the significant algorithmic advances of the last few years. While the doubly exponential bound established in [PR89] suggests that challenges to scalability will persist, there is increasing hope that synthesis algorithms may be applied to the design and diagnosis of intricate, safety-critical protocols.

Besides the synthesis of reactive systems, **functional synthesis** is concerned with extracting an
implementation—often a software program—from an input-output relation. The synthesized program reads the input and, after some time, writes the result of its computation. (Examples of such tasks include sorting an array of numbers, performing a search in a database, computing a mathematical formula.) In recent years, many domain specific automatic synthesis approaches have been proposed [MW71, SLRBE05, VYY10, Gul10]. For instance, one that has been particularly popular is “sketching” [SLRBE05, SLTB+06]. This approach is based on the idea that the designer provides the skeleton of the implementation along with its specification and the automatic synthesis approach fills in the necessary details, which results in a functionally correct implementation. The focus of this thesis is the synthesis of finite reactive systems that are implemented in hardware. Therefore, details of functional synthesis and software synthesis are not discussed, although the approaches discussed in this thesis can be used for implementing reactive systems in software and some of the approaches discussed can be adapted for functional synthesis.

There are two aspects to judge the usefulness of an automatic synthesis approach: efficiency of the realizability check and quality of the implementation. The efficiency of the realizability check is the ability of the approach to prove that the given specification is realizable or not. When the specification is proven to be realizable then the quality of the implementation is the ability of the approach to generate a cost-effective implementation. The second metric is of extreme importance when the specification describes a hardware system. Unfortunately, state-of-the-art techniques generate implementations that are simply too large to be cost-effective. This thesis describes a framework through which hardware controllers can be synthesized. Given a specification of a controller, the framework allows one to generate an implementation that is an order of magnitude smaller than implementations generated by other synthesis technique. Furthermore, in most cases the amount of time required by the synthesis framework is significantly less than other approaches.

1.1 Background

This section briefly describes the historical development of game-based approach to synthesis of a reactive system from a set of \( \omega \)-regular properties. The developments that are directly related to the contributions of this thesis are discussed in more detail in Chapter 3.
A reactive system’s intended behavior may be described by several simple properties, each given as either a formula in linear temporal logic (LTL) or as an \( \omega \)-regular automaton. In the original approach of Pnueli and Rosner, all the formulae and automata are reduced to one deterministic Rabin automaton. A deterministic Rabin game is derived from this automaton. This approach suffers from the high cost of determinization [Saf89, Löd99], which is prohibitive for even moderate-sized automata. Thanks to the development of a better determinization approach [Pit06], parity automata are now preferred over Rabin automata. It is discussed in detail in Section 3.7 that nondeterministic \( \omega \)-automata cannot always be used for synthesis and determinization of the automata is often necessary for games.

The approaches of [HRS05, HP06] avoid determinization. The approach of [HRS05] can only solve a subset of Rabin games. This implies that not all finite reactive systems can be synthesized through this approach. Therefore, its effectiveness is limited in the treatment of all \( \omega \)-regular specifications. The approach of [HP06] replaces the expensive determinization step with a less expensive step to create nondeterministic automata which are good for games. These nondeterministic automata may be larger than their determinized counterparts and their acceptance conditions is equally complex. Therefore, the advantage gained by a simpler determinization procedure may not contribute to an overall gain in the efficiency of the synthesis procedure. When the specification only contains LTL formulae then more efficient determinization procedures can be used [Sch04, MSL08, MS10].

The restriction of specifications to a subset of \( \omega \)-regular properties known as General Reactivity (1) [PPS06] has gained popularity. A GR(1) specification allows one to use algorithms more efficient than the general ones. Synthesis of such a specification is achieved by converting it to a Streett (1) game. It is claimed that in practice most reactive systems can be described by a GR(1) specification. However in practice, deep understanding of LTL and automata theory is often required to write a specification of a reactive system that is GR(1).

Alternatively, the approach of [KV05, KPV06, FJR09, FJR10, Ehl10] completely avoids determinization through alternate constructions commonly known as “Safraless” constructions. Instead of converting the specification to a two-player game, it is converted to a tree automaton; if the language of this automaton is not empty then the specification is realizable. Any word in the language of this automaton yields
an implementation. The approach of [FJR09, FJR10] builds upon the Safraless construction by generating a sequence of two-player safety games from the tree automaton\(^1\). These games are then solved using an algorithm based on antichains. The approach of [Ehl10] restricts the number of counters in the safety games at the expense of a more complex transition function of the counters. This approach then solves the safety games using a traditional algorithm for solving games.

While the above mentioned approaches only deal with specifications of a centralized reactive system, [SF07] extends the approach of [KV05, KPV06] to distributed reactive systems. As long as the specification describes controllers that share all the information with each other, the synthesis problem remains decidable. The proposed approach reduces the language emptiness check of the tree automaton to a sequence of satisfiability (SAT) problems. This approach laid the foundation for [FJR09, FJR10, Ehl10].

Despite all the recent developments in checking the realizability of the specification, the ability to generate good enough implementations has been elusive. The approach of [SHB12] addresses this issue by being careful in computing the winning strategy relation when the specification is GR(1). On the other hand, the approach [EKH12] focuses on extracting a modular strategy from the winning strategy relation; this approach is not restricted to a GR(1) specification. Both approaches rely on the efficiency of the translation of the specification to a game.

### 1.2 Thesis Contributions

This thesis deals with both aspects of the automatic synthesis procedure: efficiency of the realizability check and the quality of the implementation generated by the procedure when the specification is realizable. An important contribution of this thesis is that a synthesis tool needs not sacrifice efficiency in order to synthesize from specifications that are not syntactically or even semantically of the GR(1) type. This freedom may prove to be helpful in popularizing the automatic synthesis approach among designers who are not proficient in writing logical specifications describing the behavior of the desired reactive system.

The contributions of this thesis are summarized below:

\(^1\) By adding non-decreasing counters, one can convert the complex acceptance condition of the tree automaton to a safety acceptance condition. This is similar to the approach of Bounded Model Checking.
• **Hierarchy based approach:** Solving the game is the most expensive phase of the overall synthesis process. The size and the complexity of the game directly contribute to the amount of time spent in solving the game. I have developed an approach that interleaves the translation process with the solution of the game; this allows me to keep the game small by identifying some winning/losing states during the translation process. The interleaving depends on the temporal hierarchy of properties in the specification.

The temporal hierarchy of $\omega$-regular properties is described in Chapter 3. The safety properties are the simplest of the $\omega$-regular properties. In practice, the specification of a reactive system contains a large percentage of safety properties and the hierarchy based approach exploits this fact. Furthermore, one can identify some of the protagonist’s losing states in the game by solving a game with simpler objectives. These aspects of the hierarchy based approach significantly impact the efficiency of solving the synthesis game in practice.

• **Streett(1) and Streett($n$) Games:** The specialized synthesis approach of a GR(1) specification is the first synthesis procedure that was able to synthesize a complex specification derived from a real-world reactive system [BGJ+07a]. This approach converts GR(1) specification to a Streett(1) game and then solves it efficiently by evaluating a $\mu$-calculus formula. I show that a GR($n$) specification can be converted to Semi-Generalized Streett($n$) game which has special properties that can be exploited when solving such a game. Furthermore, I have shown that an iterative algorithm can solve a Semi-Generalized(1) game playing a sequence of Streett games with only a single Streett pair. This significantly improves the efficiency of synthesizing a GR(1) specification.

Prior to the contributions of this thesis, no efficient synthesis procedure for a GR($n$) specification has been reported. The existing approaches are unable to exploit the special nature of a game obtained from a GR($n$) specification. I have shown that the iterative approach to solving a Semi-Generalized Streett(1) game obtained from a GR(1) specification can be extended to efficiently solve the Semi-Generalized Streett($n$) obtained from a GR($n$) specification.

• **Transition Structure of the Game:** In practice, a specification is often a collection of several
small properties. Each property either describes what the system must never do (safety properties) or what the system must do under certain conditions (progress properties). In either case, each property is converted to an $\omega$-automaton that is composed with the other property automata. A game is then derived from the product automaton. The implementation derived from the winning strategy of the protagonist must follow the game’s transition structure.

In almost all the current synthesis approaches, the game’s transition structure is directly inherited by the implementation. With this in mind, the transition structure of the game is carefully constructed from the safety properties in the specification (as these are mainly instrumental in defining the transition structure of the game). This result in increasing efficiency of solving the game while simultaneously generating a simpler implementation when the specification is proven to be realizable.

- **Input-Based Games:** Due to the positive impact of symbolic algorithms, the size of input models that can be subjected to automatic synthesis has increased dramatically. Symbolic algorithms have to be applied carefully unless the benefits are nullified. The number of symbolic variables negatively affects the effectiveness of these algorithms. Until recently, the specification has always been converted to a turn-based game. I made the observation that turn-based games use more symbolic variables than necessary, and developed the theory of input-based games (which use fewer symbolic variables). I proved that key algorithms crafted for solving turn-based games can be adapted to solve input-based games.

- **General Reactivity ($n$):** Despite the popularity of GR(1) specifications, there are cases when a GR(1) specification cannot describe a reactive system. I have studied one such example and I show how my approach handles this specification.

These contributions are closely linked with each other. This will become more evident when these are discussed in detail in later chapters.
1.3 Thesis Organization

The organization of this thesis is as follows:

Chapter 2 covers the definitions related to linear time logic (LTL), $\omega$ word automata and two-player games that are pertinent to my work. It also contains a brief discussion of the symbolic representations of automata and games through reduced ordered Binary Decision Diagrams (BDDs).

Chapter 3 defines the problem of synthesis of a reactive system from an $\omega$-regular specification. It lays the foundations and discusses the high level choices I have made in developing the approach presented in this thesis. It presents a detailed discussion on why any non-deterministic $\omega$-automata are not suitable for automatic synthesis. The merits of various determinization procedures are discussed along with synthesis techniques that avoid the determinization step.

Chapter 4 discusses the details of my hierarchy-based approach. The objective of the translation of specification to automata is to obtain a two-player game; the game is then used to prove the realizability of the specification. The hierarchy-based approach exploits the temporal hierarchy of properties to generate a game that is significantly smaller and simpler to solve. I also discuss the iterative algorithm that speeds up the solution of a Streett(1) game and offers more opportunities to generate a smaller implementation.

In Chapter 5, first I discuss the challenges in generating a small implementation that have not been addressed by the hierarchy-based approach. The focus of this chapter is $\omega$-regular safety properties. As stated earlier, these properties form the bulk of most specifications of real-world reactive systems. I observe that by converting each property to an automaton and then composing all the property automata, the resulting product automaton ends up with a significant amount of redundancy. This redundancy is often difficult to remove in the later stages of the synthesis process, so that it finally gets inherited by the implementation. This is one of the main reasons why the quality of the implementation produced by state-of-the-art tools is poor. My approach extracts transition constraints from the safety automaton and then solves a safety game without the explicit construction of the product automaton. I also discuss possible extensions of this approach to progress properties.

Chapter 6 presents the theory of input-based games. It is argued that when properties are converted
to automata then the interpretation of the automata as input-based games is quite natural. This is because input-based games and synchronous hardware are closely related. The objective of the synthesis approach in this thesis is to generate a hardware implementation; therefore, input-based games are preferred over turn-based games. I establish the correspondence between input-based games and turn-based games, and thus prove that any game-solving algorithm that relies on attraction computations can also be used to solve input-based games.

Chapter 7 presents a simplified GR(1) specification inspired by the AMBA shared-bus architecture. It is then argued that if this architecture is extended to allow multiple communication channels (partial crossbar architecture) to be active at the same time then no GR(1) specification can describe such an architecture. It is shown that the specification of a full crossbar architecture can be represented by a Gr(1) formula but the specification of a partial crossbar architecture cannot be represented by an GR(1) formula.

Chapter 8 presents the results of the synthesis tool based on the approaches discussed in this thesis. This tool has been implemented within the VIS framework [B^+96]. Chapter 9 includes the conclusions of this thesis. It also discusses some future research directions.
Chapter 2

Automata, Logic and Games

2.1 Finite Transition Structures, Automata and Transducers

A finite transition structure (FTS) is a device which has finite states and accepts an input word from a finite alphabet. The FTS starts in a certain state and after reading a letter of the input word, it moves to another state in accordance with its transition function. Two types of finite transition structures are of interest here; automata and transducers. An automaton is an FTS that accepts or rejects the input word; while a transducer is an FTS that for every input letter generates an output letter selected from an output alphabet.

Both types of FTS can be used for languages consisting of either finite or infinite words. This thesis deals with languages made up of the latter. An infinite-word language that can be recognized by a finite automaton is an $\omega$-regular language. Just as finite automata and regular expressions accept regular languages [Kle56], $\omega$-automata and $\omega$-regular expressions accept $\omega$-regular languages [Büc62].

An $\omega$-automaton is a finite transition structure that accepts an $\omega$-regular language.

Definition 2.1.1. An $\omega$-automaton is a tuple $A = \langle \Sigma, Q, q_{in}, \delta, F \rangle$, where $\Sigma$ is the input alphabet, $Q$ is the finite, non-empty set of states, $q_{in} \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, and $F \subseteq Q^\omega$ is the acceptance condition.

If $\exists q \in Q. \exists \sigma \in \Sigma. \delta(q, \sigma) = \emptyset$ then the automaton $A$ is incomplete. A special type of incomplete automata are discussed in Chapter 5. Since $F$ may be an infinite set, various finite representations of $F$ are used in practice; these are discussed in Section 2.3. An $\omega$-automata reads an infinite word and generates a
run. The run of an automaton is an infinite sequence of its states.

**Definition 2.1.2.** A run $\rho$ of an automaton $A = (\Sigma, Q, q_{in}, \delta, F)$ on the infinite word $w = w_0w_1 \ldots$ where $w \in \Sigma^\omega$, is an infinite sequence of states $\rho = \rho_0\rho_1 \ldots$ where $\rho \in Q^\omega$, such that:

$$\rho_0 = q_{in} \text{ and } \forall i > 0 . \rho_{i+1} \in \delta(\rho_i, w_i) .$$

The language accepted by $\omega$-automaton $A$ is a subset of $(\Sigma)^\omega$ denoted by $L(A)$. An input word $w \in \Sigma^\omega$ belongs to $L(A)$ if there exists a run $\rho$ of $A$ on $w$ that belongs to $F$. Transducers are finite transition structure with outputs that generate an output based on given input and state.

**Definition 2.1.3.** A transducer is a tuple $M = (\Sigma_e, \Sigma_s, Q, q_{in}, \delta, O)$, where $\Sigma_e$ is the input alphabet, $\Sigma_s$ is the output alphabet, $Q$ is the finite, non-empty set of states, $q_{in} \in Q$ is the initial state, $\delta : Q \times \Sigma_e \rightarrow 2^Q$ is the transition function, and $O : Q \times \Sigma_e \rightarrow \Sigma_s$ is the output function.

As is the case for an $\omega$-automaton, a transducer reads an infinite word and generates a run, which is an infinite sequence of its states.

**Definition 2.1.4.** A run $\rho$ of a transducer $M = (\Sigma_e, \Sigma_s, Q, q_{in}, \delta, O)$ on the infinite word $w = w_0w_1 \ldots$ where $w \in \Sigma^\omega$, is an infinite sequence of states $\rho = \rho_0\rho_1 \ldots$ where $\rho \in Q^\omega$, such that:

$$\rho_0 = q_{in} \text{ and } \forall i > 0 . \rho_{i+1} \in \delta(\rho_i, w_i) .$$

A transducer reads a word from the input language $(\Sigma_e)^\omega$ and produces an output word from the output language $(\Sigma_s)^\omega$. The language of the transducer is denoted by $L(M)$ and it is a subset of $(\Sigma_e \times \Sigma_s)^\omega$. A word $(u_0, v_0)(u_1, v_1)(u_2, v_2) \ldots$ belongs to $L(M)$ if the input word $u_0u_1u_2 \ldots$ produces a run $\rho$ of $M$ such that:

$$\forall i \geq 0 . v_i = O(\rho_i, u_i) .$$

A reactive system is one that continuously interacts with its environment. A transducer is a model of a reactive system, as it continuously interacts with the environment through the input alphabet $\Sigma_e$ and the output alphabet $\Sigma_s$. Definitions 2.1.1 and 2.1.3 describe a nondeterministic FTS; there exists a special case
of FTS which is deterministic. An FTS with an input alphabet $\Sigma$ is deterministic if the following condition holds:

$$\forall q \in Q . \forall \sigma \in \Sigma . |\delta(q, \sigma)| = 1 . \quad (2.1)$$

In this thesis, the synthesis process manipulates an automaton $A$ derived from a set of properties over the alphabet $\Sigma_e \times \Sigma_s$ and constructs a deterministic transducer $M$ with input alphabet $\Sigma_e$ and an output alphabet $\Sigma_s$ such that $L(M) \subseteq L(A)$.

### 2.1.1 Transducers and Hardware

In this thesis, the objective of the synthesis process is to obtain a hardware implementation of $M$. Hardware implementations commonly rely on binary systems. The input alphabet is encoded by a set of binary inputs $X_e$ where $\Sigma_e = 2^{X_e}$, while the output alphabet is encoded by a set of binary outputs $X_s$ where $\Sigma_s = 2^{X_s}$. The set of states $Q$ is also encoded using binary state variables. The encoding of the automaton plays an important role in Chapter 6.

In a hardware implementation of a transducer, transitions are synchronized by the clock. At each clock cycle, the transducer consumes an input letter, moves into a new state and produces a new output letter. The type of hardware implementation of a transducer depends on its output function. The two basic types are discussed below:

- **Moore Machine**: The output function $O : Q \rightarrow \Sigma_s$ of a Moore machine depends only on the current state.

- **Mealy Machine**: The output function $O : Q \times \Sigma_e \rightarrow \Sigma_s$ of a Mealy machine depends on the input and the current state.

A Moore machine has to wait for the next clock cycle before it can react to an input, because its outputs are only a function of its states. On the other hand, a Mealy machine can react to an input in the same clock cycle. When it is not necessary for a transducer $M$ to react to an input in the same clock cycle then a Mealy machine can be converted to Moore machine; in such circumstances the Moore model of $M$ will often have more states than the Mealy model of $M$. 
While the Mealy model is more general, the Moore model is also of significant importance as machines based on both models are used in practice. One of the distinguishing features is that when connecting two Mealy machines with each other, the timing analysis of the product machines has to be performed again and care must be taken to avoid the creation of combinational loops. On the other hand, a Mealy model must be used when the machine must react to the inputs without any delay, which is beyond the capabilities of the Moore model. The following example describes a transducer that can only be implemented in hardware using a Mealy model.

Example 2.1.5. Consider a transducer $M = (\Sigma_e, \Sigma_s, Q, q_{in}, \delta, O)$, where:

$$
\begin{align*}
\Sigma_e &= \{i, d, s\} \\
\Sigma_s &= \{o, n\} \\
Q &= \{q_0, q_1, q_2\} \\
qu_{in} &= q_0 \\
\delta(q_k, i) &= q_j & \text{where } j = (k + 1) \mod 3 \\
\delta(q_k, d) &= q_j & \text{if } (k > 0) \text{ then } j = (k - 1) \text{ else } j = 0 \\
\delta(q_k, n) &= q_k \\
O(q_k, \sigma) &= x & \text{if } (k = 2 \text{ and } \sigma = i) \text{ then } x = n \text{ else } x = n .
\end{align*}
$$

The state of $M$ changes when the input alphabet is either $i$ or $d$, while the output is always set to $n$ unless the machine is in state $q_2$ and the input is $i$. The machine $M$ is an example of Mealy machine model.

2.2 Alternating and Universal Automata

The synthesis approach presented in this thesis is only concerned with nondeterministic and deterministic automata which are special cases of alternating automata. When comparing this approach with other synthesis approaches, the alternating and universal automata need to be defined. Universal automata are used in the “Safraless decision procedures” discussed in Section 3.10.

For a given set $Q$, let $B(Q)$ be the set of Boolean formulae over $Q$. (A Boolean formula is constructed...
from elements in $Q$ using the $\land$, $\lor$ and $\neg$ connectives.) Let $B^+(Q)$ be the set of positive Boolean formulae over $Q$. (A positive Boolean formula is a Boolean formula where the $\neg$ connective is not used.) Let $R \subseteq Q$; then $R$ satisfies a formula $F \in B^+(Q)$ if $F$ is true when the elements in $R$ are assigned $\top$ and the elements in $Q \setminus R$ are assigned $\bot$. The following example illustrates these concepts.

**Example 2.2.1.** Let $Q = \{q_1, q_2, q_3, q_4\}$. Let $F = \{(\neg q_1 \lor q_2) \land (q_3 \lor \neg q_4)\}$ be a Boolean formula and let $F^+ = (q_1 \lor q_2) \land (q_3 \lor q_4)$ be a positive Boolean formula. Consider the following subsets of $Q$:

- $R_1 = \{q_1, q_2\}$
- $R_2 = \{q_1, q_3\}$
- $R_3 = \{q_1, q_4\}$
- $R_4 = \{q_2\}$.

Then $R_1$ and $R_3$ do not satisfy $F$ while $R_2$ and $R_4$ satisfy $F$. Likewise, $R_1$ and $R_4$ do not satisfy $F^+$ while $R_2$ and $R_3$ satisfy $F^+$.

Alternating automata that read infinite words from a finite alphabet $\Sigma$ are defined as follows:

**Definition 2.2.2.** An alternating automaton on infinite words is a tuple $A = (\Sigma, Q, q_{\text{in}}, \delta, F)$, where $\Sigma$ is the input alphabet, $Q$ is the finite, non-empty set of states, $q_{\text{in}} \in Q$ is the initial state, $\delta : Q \times \Sigma \to B^+(Q)$ is the transition function, and $F \subseteq Q^\omega$ is the acceptance condition.

An alternating automaton in a state $q \in Q$ reads an input letter $\sigma \in \Sigma$ then chooses a set $R \subseteq Q$ that satisfies the positive Boolean formula $\delta(q, \sigma)$. It spawns $|R|$ parallel versions of itself, where the next state of each version is a unique state from the set $R$. The run of an alternating automaton is a leafless tree where each node of the tree is labeled by some state of the alternating automaton.

**Definition 2.2.3.** A run $T$ of the alternating automaton $A = (\Sigma, Q, q_{\text{in}}, \delta, F)$ on the infinite word $w = w_0w_1w_2\ldots$ where $w \in \Sigma^\omega$, is a leafless labeled tree such that:

- $T(\epsilon) = q_{\text{in}}$,
- every node in $T$ has $k \leq |Q|$ children,
• let the set $R \subseteq Q$ be the labels of the children of a node $n$ in $T$; then $R$ is a satisfying assignment for $\delta(q, w_i)$, where $n$ is at a depth $i$ from the root node $\epsilon$.

The word $w$ belongs to $L(A)$ if there exists a run $T$ of $A$ where every infinite path $\rho$ in $T$ belongs to the acceptance condition $F$.

Notice that $B^+_\vee(Q) \subseteq B^+(Q)$ and $B^+\wedge(Q) \subseteq B^+(Q)$, where $B^+_\vee(Q)$ is the set of positive Boolean formulae constructed from elements in $Q$ using the $\vee$ connective only and $B^+\wedge(Q)$ is the set of positive Boolean formulae constructed from elements in $Q$ using the $\wedge$ connective only. The sets $B^+_\vee(Q)$ and $B^+\wedge(Q)$ can be used to describe nondeterministic automata and universal automata as special cases of alternating automata.

**Definition 2.2.4.** An alternating automaton $A = \langle \Sigma, Q, q_{in}, \delta, F \rangle$ is a universal automaton when $\delta : Q \times \Sigma \rightarrow B^+\wedge(Q)$.

**Definition 2.2.5.** An alternating automaton $A = \langle \Sigma, Q, q_{in}, \delta, F \rangle$ is a nondeterministic automaton when $\delta : Q \times \Sigma \rightarrow B^+\vee(Q)$.

Notice that the Definition 2.1.1 can be also used to describe a universal automaton provided the definition of run is changed. In this thesis, however, the $\omega$-automata under consideration are nondeterministic unless otherwise specified. As mentioned in Section 2.1, a run of a nondeterministic automaton is a leafless tree with only one path. An example of a nondeterministic automaton and an example of a universal automaton are shown in Figure 3.14.

Complementation of alternating automata can be achieved by dualization. Given an alternating automaton $A = \langle \Sigma, Q, q_{in}, \delta, F \rangle$ one can construct an alternating automaton $A' = \langle \Sigma, Q, q_{in}, \delta', F' \rangle$ which accepts the complement language $\Sigma^\omega \setminus L(A)$, where $\delta'(q, \sigma)$ is obtained from $\delta(q, \sigma)$ by replacing $\wedge$ with $\vee$ and $\vee$ with $\wedge$ connectives, and the acceptance condition $F' = Q^\omega \setminus F$. When $A$ is a nondeterministic (universal) automaton then $A'$ is a universal (nondeterministic) automaton. Similarly, when $A$ is a deterministic automaton then $A'$ is also a deterministic automaton. The procedure for the complementation of $F$ depends on its representation and is discussed in Section 2.3.
The discussion until now has been about \( \omega \)-automata that read infinite words. In addition, there exist \( \omega \)-automata that read infinite trees. An infinite tree is a leafless tree annotated with the letters from a finite alphabet \( \Sigma \). Although, this thesis is not concerned with tree automata, there is a connection between tree automata and games [KV05, KPV06]. The synthesis approach based on “Safraless decision procedures” discussed in Section 3.10 uses tree automata to solve a synthesis game. In this thesis, the input to \( \omega \)-automata is assumed to be infinite words unless otherwise specified.

2.3 Omega Automata

The set \( F \subseteq Q^{\omega} \) described in the definition of an \( \omega \)-automaton may be an infinite set. An acceptance condition \( \alpha \) can be defined to effectively represent the set \( F \). Several acceptance conditions are used in the literature; Büchi [Büc62], co-Büchi, Rabin [Rab69], Streett [Str81], parity [Mos84, EJ91], and Muller [Mul63] are described in this section. These acceptance conditions are about the set of states \( \inf(\rho) \) that occur infinitely often in a run \( \rho \).

**Definition 2.3.1.** A run \( \rho \) is accepting for a Büchi acceptance condition \( B \subseteq Q \) iff \( \inf(\rho) \cap B \neq \emptyset \).

**Definition 2.3.2.** A run \( \rho \) is accepting for a co-Büchi acceptance condition \( C \subseteq Q \) iff \( \inf(\rho) \cap C = \emptyset \).

**Definition 2.3.3.** A run \( \rho \) is accepting for a Rabin acceptance condition \( R = \{(U_1, E_1), \ldots, (U_n, E_n)\} \subseteq 2^Q \times 2^Q \) iff for some \( i \) we have \( \inf(\rho) \cap U_i \neq \emptyset \) and \( \inf(\rho) \cap E_i = \emptyset \).

**Definition 2.3.4.** A run \( \rho \) is accepting for a Streett acceptance condition \( S = \{(U_1, E_1), \ldots, (U_n, E_n)\} \subseteq 2^Q \times 2^Q \) iff for all \( i \) we have \( \inf(\rho) \cap U_i = \emptyset \) or \( \inf(\rho) \cap E_i \neq \emptyset \).

**Definition 2.3.5.** A (nongeneralized) parity acceptance condition assigns a priority to each state of the automaton. Letting \( [k] = \{i \mid 0 \leq i < k\} \), a parity condition of index \( k \) is a function \( \pi : Q \rightarrow [k] \). A run \( \rho \) is accepting iff \( \max\{\pi(q) \mid q \in \inf(\rho)\} \) is odd; that is, iff the highest recurring priority is odd.

When the context is clear a nongeneralized parity condition is also referred to as a parity condition. In the literature, one finds different conventions used to define parity conditions, which use lowest recurring
color or swap the roles of even and odd colors. To relate those different definitions, a new parity function $\pi'$ can be obtained from $\pi$; the acceptance of a run $\rho$ for $\pi'$ can be defined as:

- A run $\rho$ is accepting iff $\max\{\pi'(q) \mid q \in \inf(\rho)\}$ is even where $\pi'(q) = \pi(q) + 1$.
- A run $\rho$ is accepting iff $\min\{\pi'(q) \mid q \in \inf(\rho)\}$ is odd where $\pi'(q) = k - \pi(q)$.
- A run $\rho$ is accepting iff $\min\{\pi'(q) \mid q \in \inf(\rho)\}$ is even where $\pi'(q) = k - \pi(q) - 1$.

These definitions of the parity condition defined above are therefore all equivalent. Definition 2.3.5 is the only one used in this thesis.

Given a set of parity conditions $\mathcal{P} = \{\pi_1, \pi_2, \ldots, \pi_k\}$, a generalized parity condition is a Boolean formula $\Pi_g \in \mathcal{B}(\mathcal{P})$; a run $\rho$ is accepting if there exists a satisfying assignment $P \subseteq \mathcal{P}$ of the Boolean formula $\Pi_g$ such that $\rho$ is accepting for every $\pi \in P$ and rejecting for every $\pi \in (\mathcal{P} \setminus P)$. A disjunctive parity condition is a formula $\Pi_d \in \mathcal{B}_+^\vee(\mathcal{P})$; a run $\rho$ is accepting for a disjunctive parity condition $\Pi_d$ iff it is accepting for some $\pi \in \mathcal{P}$ such that $\pi = \top$ satisfies the formula $\Pi_d$. A conjunctive parity condition is a formula $\Pi_c \in \mathcal{B}_+^\wedge(\mathcal{P})$; let $P \subseteq \mathcal{P}$ be the set of parity conditions appearing in the formula $\Pi_c$; then a run $\rho$ is accepting for $\Pi_c$ iff it is accepting for all the parity conditions $\pi \in P$.

**Definition 2.3.6.** A run $\rho$ is accepting for a Muller acceptance condition $\mathcal{M} \subseteq 2^Q$ iff $\inf(\rho) \in \mathcal{M}$.

Muller automata are the most versatile because an automaton $A = \langle \Sigma, Q, q_{in}, \delta, \alpha \rangle$ with another type of acceptance condition $\alpha$ can be converted to a Muller automaton $A_M = \langle \Sigma, Q, q_{in}, \delta, \mathcal{M} \rangle$, that is, without changing the transition structure. Despite this quality, Muller acceptance conditions are not commonly used in practice because the number of elements in $\mathcal{M}$ increases exponentially with the number of states. Generalized parity automata are equally versatile, but the full generality of a parity acceptance condition is only required when converting an automaton $A$ with a Muller acceptance condition to a generalized parity automaton $A_P = \langle \Sigma, Q, q_{in}, \delta, \Pi_g \rangle$ without changing the transition structure. On the other hand, $A$ with a non-Muller acceptance condition can always be converted to a parity automaton $A_P$ with a conjunctive, disjunctive or even nongeneralized parity condition as shown later in this section. It will soon be seen that
parity conditions are easily complemented. Another important advantage of parity acceptance conditions in context of determinization of a nondeterministic Büchi automaton is discussed in Section 3.7.

[BK10] discusses the conversion of one acceptance condition to another without changing the structure of the automaton. However parity automata are excluded from that discussion. Here the conversion of deterministic Büchi, co-Büchi, Rabin and Streett automata to a conjunctive or disjunctive parity automata is given. First, complementation of a nongeneralized parity automaton is defined. Given a parity automaton \( A = (\Sigma, Q, q_{\text{in}}, \delta, \pi) \) then negation of \( \pi \) denoted as \( \pi' \) is defined as:

\[
\pi'(q) = \begin{cases} 
\pi(q) - 1 & \forall q \in Q : \pi(q) \neq 0 \\
\pi(q) + 1 & \text{otherwise.}
\end{cases}
\]  

(2.2)

The negation of a conjunctive (disjunctive) parity condition is a disjunctive (conjunctive) parity condition where all the conjuncts (disjuncts) are negated. A Büchi condition can be converted to a parity condition as follows:

\[
\pi(q) = \begin{cases} 
1 & q \in B \\
0 & \text{otherwise.}
\end{cases}
\]  

(2.3)

A co-Büchi condition can be converted to a parity condition as follows:

\[
\pi(q) = \begin{cases} 
2 & q \in C \\
1 & \text{otherwise.}
\end{cases}
\]  

(2.4)

A Rabin condition can be converted to a disjunctive parity condition as follows:

\[
\pi_i(q) = \begin{cases} 
2 & q \in E_i \\
1 & q \in U_i \\
0 & \text{otherwise.}
\end{cases}
\]  

(2.5)

A Streett condition can be converted to a conjunctive parity condition as follows:

\[
\pi_i(q) = \begin{cases} 
3 & q \in E_i \\
2 & q \in U_i \\
1 & \text{otherwise.}
\end{cases}
\]  

(2.6)
A parity acceptance condition is also known as **Rabin chain** condition because it is a special case of Rabin acceptance condition. Let $A = (\Sigma, Q, q_{in}, \delta, \pi)$ be a parity automaton and let $S = \{S_0, S_1, \ldots, S_{k-1}\}$ be a partition of $Q$ such that $\forall s \in S_i. \pi(s) = i$. The following chain of states can be formed:

\[
Q_0 \subset Q_1 \subset Q_2 \subset \ldots Q_{k-2} \subset Q_{k-1} , \quad \text{where} \quad \\
Q_0 = S_0 \\
Q_1 = S_0 \cup S_1 \\
\ldots \\
Q_{k-1} = \bigcup_{0 \leq i < k} S_i .
\]

Notice that $Q = Q_{k-1}$. A run $\rho$ is accepted by $A$ iff

\[
\forall 0 \leq i < j < k . Q_i \subset \inf(\rho) \text{ and } \inf(\rho) \subseteq Q_j \text{ and } j \text{ is odd} .
\]

The Rabin acceptance condition $R$ for $A$ when $k$ is even, is given by

\[
R = \{(S_1, Q \setminus Q_1), (S_3, Q \setminus Q_3), \ldots (S_{k-1}, \emptyset)\} ,
\]

otherwise,

\[
R = \{(S_1, Q \setminus Q_1), (S_3, Q \setminus Q_3), \ldots (S_{k-2}, Q \setminus Q_{k-2})\} .
\]

The connection of parity acceptance condition and Rabin acceptance condition is further discussed in Section 3.7.

The parity automaton can also be equipped with a Streett condition (which is the dual of a Rabin condition). Since the negation of a parity condition is another parity condition and any parity automaton can be equipped with a Rabin condition, therefore, it can also be equipped with a Streett condition. A parity condition can be converted to a Streett condition directly without the need to convert it to a Rabin condition first. Let $A = (\Sigma, Q, q_{in}, \delta, \pi)$ be a parity automaton and let $S = \{S_0, S_1, \ldots, S_{k-1}\}$ be a partition of $Q$ such that $\forall s \in S_i. \pi(s) = i$. The Streett acceptance condition $S$ for $A$ when $k$ is even, is given by

\[
S = \{(S_0, S_1 \cup S_3, \ldots S_{k-1}), (S_2, S_3 \cup S_5, \ldots S_{k-1}), \ldots (S_{k-2}, S_{k-1})\} ,
\]
otherwise, 
\[ S = \{(S_0, S_1 \cup S_3, \ldots S_{k-2}), (S_2, S_3 \cup S_5, \ldots S_{k-2}), \ldots (S_{k-1}, \emptyset)\}. \]

The complementation of an alternating automaton \( A = (\Sigma, Q, q_{\text{fin}}, \delta, F) \) is discussed in Section 2.2, which is revisited here. Given that the infinite set \( F \) is represented by an acceptance condition \( \alpha \) then \( \Sigma^\omega \setminus F \) is represented by another acceptance condition \( \alpha' \). If \( \alpha \) is a Büchi (co-Büchi) acceptance condition then \( \alpha' \) is a co-Büchi (Büchi) acceptance condition, similarly, if \( \alpha \) is a Rabin (Streett) acceptance condition then \( \alpha' \) is a Streett (Rabin) acceptance condition. On the other hand, if \( \alpha \) is a parity acceptance condition then \( \alpha' \) is another parity acceptance condition respectively. The complementation of \( \alpha \) to obtain \( \alpha' \) can be understood through the negation of a parity acceptance condition defined above. Finally, if \( \alpha \) is a Muller acceptance condition then \( \alpha' \) is another Muller acceptance condition where \( \alpha' = 2^Q \setminus \alpha \).

This thesis adopts the popular three-letter abbreviations to designate different types of automata [KV05, KPV06]. The first letter denotes the type transition function: A for alternating, D for deterministic, N for nondeterministic and U for universal. The second letter denotes the type of acceptance condition: B for Büchi, C for co-Büchi, M for Muller, P for parity, R for Rabin, and S for Streett. Finally, the last letter denotes the type of input language: A for language on finite words, W for language on infinite words and T for language on infinite trees. As an example, NBW is an abbreviation for nondeterministic Büchi word automaton, while DPW is an abbreviation for deterministic parity word automaton. This thesis is mainly concerned with Büchi and parity acceptance conditions.

### 2.4 Nongeneralized Parity Automata

An automaton with a generalized parity condition can be converted to another automaton with a nongeneralized parity condition. The discussion in this section is restricted to converting automata with conjunctive or disjunctive parity conditions to automata with nongeneralized parity conditions. The approaches discussed here can then be extended to converting automata with generalized parity conditions to automata with nongeneralized parity conditions.

Let \( A = (\Sigma, Q, q_{\text{fin}}, \delta, \Pi) \) be a parity automaton. Let \( \Pi \) be a conjunctive parity condition \( \Pi = \)
\{\pi_1, \pi_2, \ldots, \pi_n\} \text{ such that every condition } \pi_i \in \Pi \text{ is a Büchi acceptance condition. Then a parity automaton } A' = \langle \Sigma', Q', q'_\text{in}, \delta', \pi' \rangle \text{ can be obtained such that } L(A) = L(A'), \text{ where } \pi' \text{ is a nongeneralized Büchi acceptance condition. Let the set } C = \{c_1, c_2, \ldots, c_{n+1}\} \text{ be the memory, then the following holds:}

$$
\begin{align*}
\Sigma' & = \Sigma \\
Q' & = Q \times C \\
q'_\text{in} & = (q_\text{in}, c_1) \\
\delta'((q, c_i), \sigma) & = \\
& \begin{cases} 
(\delta(q, \sigma), c_1) & \text{if } i = n + 1 \\
(\delta(q, \sigma), c_{i+1}) & \text{if } \pi_i(q) = 1 \\
(\delta(q, \sigma), c_i) & \text{otherwise}
\end{cases} \\
\pi'((q, c_i)) & = \\
& \begin{cases} 
1 & \text{if } i = n + 1 \\
0 & \text{otherwise.}
\end{cases}
\end{align*}
$$

The set \(C\) denotes the state of a counter that resets to 1 after counting from 1 to \(n + 1\). Corollary 4.2.4, discussed in Chapter 4, shows that if \(\Pi\) is a disjunctive parity condition then it can be converted into a nongeneralized parity condition without the introduction of a counter.

Now consider the case when \(\Pi = \Pi_e \to \Pi_s\), where \(\Pi_e = \{\pi_1^e, \pi_2^e, \ldots, \pi_m^e\}\) is a conjunctive parity condition \(\pi_1^e \land \pi_2^e \land \ldots \land \pi_m^e\) such that every \(\pi \in \Pi_e\) is a Büchi acceptance condition. Similarly, \(\Pi_s = \{\pi_1^s, \pi_2^s, \ldots, \pi_n^s\}\) is another conjunctive parity condition \(\pi_1^s \land \pi_2^s \land \ldots \land \pi_n^s\) such that every \(\pi \in \Pi_s\) is a Büchi acceptance condition. The parity condition \(\Pi = \neg\Pi_e \lor \Pi_s\) is a generalized parity condition\(^1\), where \(\neg\Pi_e = \{-\pi_1^e, -\pi_2^e, \ldots, -\pi_m^e\}\) is a disjunctive parity condition \(-\pi_1^e \lor -\pi_2^e \lor \ldots \lor -\pi_m^e\), while \(\Pi = \neg\pi_1^e \lor -\pi_2^e \lor \ldots \lor -\pi_m^e \lor (\pi_1^s \land \pi_2^s \land \ldots \land \pi_n^s)\). A parity automaton \(A' = \langle \Sigma', Q', q'_\text{in}, \delta', \pi' \rangle\) can be constructed such that \(L(A) = L(A')\), where \(\pi'\) is a nongeneralized parity condition. Let the set \(C_e = \{c_1^e, c_2^e, \ldots, c_{m+1}^e\}\) and the set \(C_s = \{c_1^s, c_2^s, \ldots, c_{n+1}^s\}\) be two sets of memory, then the following

\(^1\) This parity condition is also known as a generalized Streett(1) condition [PPS06].
The special cases discussed above are important because less memory is required when a nongeneralized parity automaton is constructed through the general approach [Tho95, BLV96].

The approach of [Tho95] is now briefly described. Whenever \( \Pi \) is a set of conjunctive parity conditions where \( \pi \in \Pi \) is an arbitrary parity condition, then simple counters are not enough to convert \( \Pi \) to a nongeneralized parity condition \( \pi' \). The latest appearance records (LAR) discussed in [B"uc83, Tho95, BLV96] are used to construct a new automaton \( A' \) with a nongeneralized parity condition \( \pi' \).

One can always convert the generalized parity condition \( \Pi \) to a Muller condition \( M \). An LAR can be constructed from the automaton and \( M \), the objective of the latest appearance record is to keep track of the sequence of states that are visited infinitely often. There are two components in an LAR; an appearance record which records the sequence in which the states have been visited (in other words its value is always
one of the permutations of all the states), and a hit position which identifies the set of states that are being visited infinitely. When the token moves to another state, the value of LAR is updated, the update function is given by $P : P(Q) \times Q \rightarrow P(Q)$, where $P(Q)$ is the set of permutation of the states in $Q$. The hit position is given by the function $H : P(Q) \times Q \rightarrow \{1, 2, \ldots, n\}$ where $n = |Q|$. Given $(q_1, q_2, \ldots, q_i, \ldots q_n) \in P(Q)$ and $q_i \in Q$, then

$$P((q_1, q_2, \ldots, q_i, \ldots q_n), q_i) = (q_1, q_2, \ldots, q_{i-1}, q_{i+1}, q_n, q_i)$$

$$H((q_1, q_2, \ldots, q_i, \ldots q_n), q_i) = i .$$

In other words, the new value of the LAR is derived from the previous value by moving the state $q_i$ to the right most position all the states that were on the right of the state of $q_i$ are shifted to the left. The hit position is the index of the state $q_i$ in the given LAR. (The initial value of the LAR is arbitrarily chosen.)

The priorities from the set $\{1, 2, \ldots, n\}$ are assigned based on the number of states on the right of the hit position including the hit position. Given $(q_1, q_2, \ldots, q_i, \ldots q_n) in P(Q)$ and the hit position $i$ the priority of assigned to the LAR value $((q_1, q_2, \ldots, q_i, \ldots q_n), i)$ is either $n - i + 1$ or $n - i_2$. The former is assigned when $\{q_i, \ldots, q_n\} \in M$ and the latter is assigned when $\{q_i, \ldots, q_n\} \not\in M$. In other words, the size of the set of states which are being visited infinitely often dictates the priority of the respective value of the LAR.

**Example 2.4.1.** Consider the parity automaton shown in Figure 2.1. The Muller acceptance condition for this automaton is $\{\{A\}, \{A, B, C\}\}$. The composition of LAR with this automaton is shown in Figure 2.2.
Figure 2.2: A non-generalized parity automaton which is language equivalent to the conjunctive parity automaton of Figure 2.1. The underscore indicates the hit position. As there were no states that were assigned priority 2 or 3, the priorities 4 and 5 got bumped down to 2 and 3 respectively.

2.5 LTL: Linear Time Logic

Linear Time Logic (LTL) [WVS83, LP85] is a popular temporal logic for the specification of non-terminating reactive systems. Every LTL formula describes an $\omega$-regular language. LTL formulae are built from a set of atomic propositions, Boolean connectives, and basic temporal operators $^2$ $X$ (next), $U$ (until), and $R$ (releases). Derived operators $G$ (always) and $F$ (eventually) are usually included for convenience. An LTL formula can be converted to an $\omega$-regular automaton, which is discussed in Section 3.4.

The semantics of LTL can now be defined. Let $X$ be a set of atomic proposition and let $\Sigma = 2^X$ be the corresponding alphabet. A word $\sigma$ is an infinite sequence of letters from $\Sigma$, formally $\sigma : \mathbb{N} \rightarrow \Sigma$. Let $\sigma^i$
abbreviate $\sigma(i)$ then
$$\sigma = \sigma^0, \sigma^1, \sigma^2, \ldots,$$ (2.7)
then the suffix starting at $\sigma^i$ is denoted by
$$\sigma[^i,\infty) = \sigma^i, \sigma^{i+1}, \sigma^{i+2}, \ldots,$$ (2.8)
similarly, a segment of the word $\sigma$ from index $i$ to $j$ is denoted by
$$\sigma[^i,j] = \sigma^i, \sigma^{i+1}, \ldots, \sigma^{j-1}, \sigma^j.$$ (2.9)

Suppose $\sigma = \sigma^1, \sigma^2, \sigma^3, \ldots$ and $\varphi$ is an LTL formula defined over $\Sigma$. Let $L(\varphi)$ be the language of $\varphi$; then

- $\sigma \models T$ ,
- $\sigma \not\models \bot$ ,
- $\sigma \models \varphi$ iff $\sigma^0 \models \varphi$ when $\varphi \in X$ ,
- $\sigma \models \neg \varphi$ iff $\sigma \not\models \varphi$ ,
- $\sigma \models \varphi \land \psi$ iff $\sigma \models \varphi$ and $\sigma \models \psi$ ,
- $\sigma \models X\varphi$ iff $\sigma[^1,\infty) \models \varphi$ ,
- $\sigma \models \varphi \lor \psi$ iff $\exists i \geq 0. \sigma[^i,\infty) \models \psi$ and $\forall 0 \leq j < i. \sigma[^j,\infty) \models \varphi$ ,
- $\sigma \models \varphi \lor \psi$ iff $\forall i \geq 0. \sigma[^i,\infty) \models \varphi$ or $\exists i \geq 0. \sigma[^i,\infty) \models \psi \land \forall 0 \leq j \leq i. \sigma[^j,\infty) \models \varphi$ .

The operators $F$ and $G$ are abbreviations, where $F\varphi = \top \lor \varphi$ and $G\varphi = \bot \land \varphi$. Therefore,

- $\sigma \models F\varphi$ iff $\exists i \geq 0. \sigma[^i,\infty) \models \varphi$ ,
- $\sigma \models G\varphi$ iff $\forall i \geq 0. \sigma[^i,\infty) \models \varphi$ .

An LTL formula is in negation normal form if negation is restricted to atomic propositions.

As stated earlier, every LTL formula describes an $\omega$-regular language. However, there exist $\omega$-regular languages that cannot be described by any LTL formula. The following example illustrates this fact.
Example 2.5.1. Let $\Sigma = \{\neg a, a\}$ be the alphabet and $\varphi$ be a property which states that every odd letter of $\sigma$ must be $a$. This safety property is described by the $\omega$-regular expression $(\{\neg a, a\}, a)\omega$ and by the $\omega$-automaton shown in Fig. 2.3. There does not exists any LTL formula over the alphabet $\{\neg a, a\}$ that can describe the language $L(\varphi)$ [Wol81].

In such cases, the alphabet can be augmented with extra letters that enable the LTL formula to "count". Let $\Sigma' = \{\neg a \land \neg q, \neg a \land q, a \land \neg q, a \land q\}$ and

$$
\varphi = \neg q \land G \left( (\neg q \rightarrow X(q \land a)) \land (q \rightarrow X\neg q) \right),
$$

then the projection of $L(\varphi)$ over $\Sigma$ is $(\{\neg a, a\}, 1)\omega$.

A similar idea to the one discussed in the above example is used to convert any safety formula to a transition constraint; the procedure is discussed in Chapter 5.
2.6  Borel Hierarchy and Omega-Regular Languages

The Borel hierarchy is a classifications of Borel sets [Kec95]. The Borel sets of a topological space are defined recursively, where:

\[
G_1 = \{ A \subseteq \Sigma^\omega : A \text{ is open} \} \]

\[
F_1 = \{ A \subseteq \Sigma^\omega : A \text{ is closed} \}
\]

and \( F_1 = G_0 = G_1 \cap F_1 \), while \( F_n \) and \( G_n \) for \( n > 1 \) are defined as follows:

\[
G_{n+1} = \{ A \subseteq \Sigma^\omega : A = \bigcup_{i \geq 0} A_i \text{ where } A_i \in F_n \} \]

\[
F_{n+1} = \{ A \subseteq \Sigma^\omega : A = \bigcap_{i \geq 0} A_i \text{ where } A_i \in G_n \} .
\]

The Borel hierarchy in Figure 2.4 shows that the sets \( G_{n+1} \) and \( F_{n+1} \) strictly contain the sets \( G_{n-1} \), \( F_{n-1} \), and \( G_{n+1} \cap F_{n+1} \) from the lower levels of the hierarchy.

A topological classification of \( \omega \)-regular languages was first proposed in [Lan69]. The \( \omega \)-regular languages are contained in the set \( G_3 \cap F_3 \). The languages in this class are further divided into an infinite hierarchy based on the Rabin index of the language, which is discussed in Section 3.7. As one can observe these languages are very low level in the Borel hierarchy. In [MP90, MP95] the authors\(^3\) named \( G_1 \) as safety class, \( F_1 \) as guarantee class, \( G_2 \cap F_2 \) as obligation class, \( G_2 \) as persistence class, \( F_2 \) as recurrence class and \( G_3 \cap F_3 \) as general reactivity class. The non-safety classes \( F_1, G_1 \cap F_1, F_2, G_2, F_2 \cap G_2 \) are collectively called the progress class.

\(^3\) The clopen set \( G_1 \cap F_1 \) which is at the bottom of the hierarchy was not named.
2.6.1 Syntactic Characterization of LTL

A syntactic characterization of LTL formulae was also provided in [MP90, MP95]. A few examples of LTL formulae belonging to each class are described below.

\[ G \varphi \] is a safety formula,

\[ F \varphi \] is a guarantee formula,

\[ G K \lor F \varphi \] is an obligation formula,

\[ F G \varphi \] is a persistence formula,

\[ GF \varphi \] is a recurrence formula,

\[ \bigwedge_{1 \leq i \leq m} (GF \varphi_i \rightarrow GF \varphi) \] is a reactive\((m)\) formula,

Figure 2.4: First few levels of Borel hierarchy.
where $\mathcal{J}$, $\mathcal{K}$, $\mathcal{J}_i$, and $\mathcal{K}_i$ are past-time LTL formulae and $m$ is a natural number. The general reactivity class is layered into sub-classes denoted by general reactivity$(n)$ for $n > 0$. The sub-class general reactivity$(1)$ is of particular importance. The formulae belonging to general reactivity$(m)$ have the following form:

$$\bigwedge_{1 \leq k \leq m} \left[ \bigwedge_{1 \leq i \leq p_k} (G\ F\ \mathcal{K}_i^k) \rightarrow \bigwedge_{1 \leq i \leq n_k} (G\ F\ \mathcal{J}_i^k) \right], \quad (2.10)$$

where $\mathcal{J}_i^k$ and $\mathcal{K}_i^k$ are past-time LTL formulae and $m, p_k$ and $n_k$ are natural numbers. The formulae belonging to general reactivity$(1)$, in particular, have the following form$^4$:

$$\bigwedge_{1 \leq i \leq p} (G\ F\ \mathcal{K}_i) \rightarrow \bigwedge_{1 \leq i \leq n} (G\ F\ \mathcal{J}_i). \quad (2.11)$$

As discussed in Section 2.5, there are safety properties that cannot be express by LTL directly. Similarly, there are progress properties that cannot be expressed by LTL directly. The safety class is of significant importance in practice. An important aspect of LTL formulae was discussed in [Sis94], where it was shown that LTL formulae in negation normal form that do not use the until operator define safety properties. This is a sufficient syntactic check to detect safety properties. There exist LTL formulae which define safety properties but fail the syntactic check. One can always determine if an LTL formula defines a safety property by translating it to a deterministic automaton and then computing the Rabin index of the language accepted by the automaton. This procedure is discussed briefly in Section 3.7. However, properties that fail the syntactic check are quite rare in practice. Even mildly pathological cases like $(p U q) \lor G\ p$ are usually recognized as safety properties by translators.

### 2.6.2 Omega Automata and Borel Hierarchy

It was shown in [McN66] that DSW and DRW are as expressive as NBW, NRW, and NSW, which express all $\omega$-regular languages. In [Lan69], it was proven that DBW are less expressive than NBW; for instance, the language described by the LTL property $F\ G\ p$ is not accepted by any DBW. The DBWs can express languages in the set $F_2$. Similarly it is proved in [LT00] that DCW and NCW are less expressive and both can express languages in the set $G_2$, for instance the language described by the LTL property $G\ F\ p$ cannot be expressed by any NCW or DCW.

$^4$ This formula can be converted to a generalized Streett$(1)$ condition.
A Büchi automaton $A = \langle \Sigma, Q, q_{\text{ini}}, \delta, B \rangle$ is weak iff every strongly connected component (SCC) either is contained in $B$ or does not intersect $B$. The definition of a weak automaton can be extended to other acceptance conditions as well. Deterministic weak $\omega$-automata accept the regular languages in $F_2 \cap G_2$ [LT00], while nondeterministic weak word automata and deterministic co-Büchi word automata are equally expressive [LT00].

### 2.7 Two-Player Games

A Gale-Stewart game is a two-player game of perfect information whose winning condition is given by a Borel set. The game is defined using a finite alphabet $\Sigma$. The two players, the antagonist (Player 0) and the protagonist (Player 1) alternate turns, and each player is aware of all the moves that have been made before making the next one. The players are also aware, from the beginning, of the condition that will determine who wins the game.

When it is its turn to play, a player chooses a single letter from $\Sigma$ and appends it to a sequence that is initially empty. The same element may be chosen more than once without restriction. The play continues without end, so that a single play of the game determines an infinite sequence of elements of $\Sigma$. The winning condition is a subset of $\Sigma^{\omega}$. If the infinite sequence created by a play of the game belongs to the winning condition, then protagonist wins, otherwise, the antagonist wins; there are no ties. It was proven in [Mar75] that these games are determinate, which is to say that one of the players will always have a winning strategy for such games. Therefore, the $\omega$-regular games (the games with a winning condition from the set $F_3 \cap G_3$) are determinate.

The Gale-Stewart games with $\omega$-regular objectives can be played on finite directed graphs, which are often bipartite. These graphs can be interpreted as specialized FTS. The nodes of the graph represent states while the directed edges represent transitions. The classical model of games is based on the notion that the next state of the game is determined by the moves of one of the player. As described earlier, games based on this notion are called turn-based games. However the objective of this thesis is to use games in designing hardware where the state changes when the clock ticks; the inputs of the hardware model are controlled by one player while the outputs are controlled by the other player. This lead to the development of input-based
games. Both types of game are defined later in this section. Both types of game rely on the same principles, and the correspondence between the two is discussed in Chapter 6. The theory developed for turn-based games can be applied to input-based games after the necessary adjustments.

**Definition 2.7.1.** A turn-based game is a tuple $G = \langle \Sigma, Q, Q_0, Q_1, q_{in}, \delta, F \rangle$, where $\Sigma$ is a finite alphabet, $Q$ is a finite non-empty set of states, partitioned into $Q_0$ (antagonist states) and $Q_1$ (protagonist states), $q_{in} \in Q$ is an initial state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function such that

$$\forall q \in Q. \exists \sigma \in \Sigma . \delta(q, \sigma) \neq \emptyset$$

and $F \subseteq Q^\omega$ is a winning condition.

In state $q \in Q_1$, Player $i$ moves to a successor $q'$ by choosing $\sigma \in \Sigma$ such that $q' \in \delta(q, \sigma)$, the two players produce an infinite sequence of states of the game known as the play of the game. When the alphabet $\Sigma$ is not required, the definition of $G$ can be simplified to obtain the classical definition of a turn-based game, the simplified turn-based game is a tuple $G' = \langle Q, Q_0, Q_1, q_{in}, \delta', F \rangle$ where $\delta' : Q \rightarrow (2^Q \setminus \emptyset)$.

In the turn-based game $G'$, in state $q \in Q$, Player $i$ moves to a successor by choosing any $q' \in \delta(q)$.

**Definition 2.7.2.** A play $\rho$ in the game $G = \langle \Sigma, Q, Q_0, Q_1, q_{in}, \delta, F \rangle$ produced by the infinite sequence of choices of the two players $w = w_0 w_1 w_2 \ldots$ where $w \in \Sigma^\omega$, is an infinite sequence of states $Q$ such that:

$$\rho_0 = q_{in} \text{ and } \forall i > 0. \rho_{i+1} \in \delta(\rho_i, w_i) .$$

A play $\rho$ in the turn-based game $G$ is won by Player 1 (protagonist) if $\rho \in F$, otherwise it is a losing play for Player 1 (winning for Player 0). In this thesis, when the play is declared winning or losing then the player being referred is always Player 1 unless otherwise specified.

**Definition 2.7.3.** An input-based game is a tuple $G = \langle \Sigma, Q, D, q_{in}, \delta, F \rangle$, where $\Sigma$ is a finite input alphabet, $Q$ is a finite non-empty set of states, $D \subseteq Q \times \Sigma$ specifies a set of allowed input letters for each state, $q_{in} \in Q$ is an initial state, $\delta : D \rightarrow (Q \setminus \emptyset)$ is a transition function such that $\forall q \in Q. \exists \sigma \in \Sigma . (q, \sigma) \in D$, and $F \subseteq Q^\omega$ is a winning condition.
In this thesis, the alphabet $\Sigma$ is the Cartesian product $\Sigma_{ed} \times \Sigma_s \times \Sigma_{ep}$ of a disclosed antagonist alphabet $\Sigma_{ed}$, a protagonist alphabet $\Sigma_1$, and a private antagonist alphabet $\Sigma_{ep}$. This definition of $\Sigma$ eventually allows the synthesis of a Moore model or a Mealy model (where the outputs are functionally dependent on some or all of the inputs). Throughout this thesis $\sigma_{ed} \in \Sigma_{ed}$, $\sigma_1 \in \Sigma_1$, $\sigma_{ep} \in \Sigma_{ep}$ and $\sigma \in \Sigma$. When the token is in state $q \in Q$, initially Player 0 chooses a letter $\sigma_{ed}$ such that

$$\exists \sigma_1, \exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_1, \sigma_{ep})) \in D,$$

and discloses it to Player 1; then Player 1 chooses a letter $\sigma_1$ such that

$$\exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_1, \sigma_{ep})) \in D,$$

and discloses it to Player 0; then Player 0 selects a letter $\sigma_{ep}$ such that $(q, (\sigma_{ed}, \sigma_1, \sigma_{ep})) \in D$; finally the token moves to $q' = \delta(q, (\sigma_{ed}, \sigma_1, \sigma_{ep}))$.

**Definition 2.7.4.** A play $\rho$ in the game $G = (\Sigma, Q, D, q_{in}, \delta, F)$ produced by the infinite sequence of choices of the two players choices $w = (t_0, u_0, v_0)(t_1, u_1, v_1)(t_2, u_2, v_2) \ldots$ where $w \in (\Sigma_{ed} \times \Sigma_s \times \Sigma_{ep})^\omega$, is an infinite sequence of states $Q$ such that:

$$\rho_0 = q_{in} \text{ and } \forall i > 0. (\rho_i, (t_i, u_i, v_i)) \in D \land \rho_{i+1} \in \delta(\rho_i, (t_i, u_i, v_i)) .$$

As was the case in turn-based games, a play $\rho$ in the input-based game $G$ is won by Player 1 (protagonist) if $\rho \in F$, otherwise it is a losing play for Player 1 (winning for Player 0). The importance of the set $D$ which is a state and input-letter pair is evident in the definition of subgames of input-based games discussed in Chapter 6. It is important to note that the winning states of each player in these games can be determined without considering the initial state: the player whose winning states contain the initial state is declared the winner. Therefore, when it is convenient (for instance, in Chapter 6) the description of either type of game may omit the initial state.

The following example shows the input-based game and the turn-based game obtained from an LTL property. In the figures of this thesis depicting turn-based games, squares represent the states that are controlled by Player 0 (antagonist) and circles represent the states that are controlled by Player 1 (protagonist).
Example 2.7.5. Let $\varphi = G(\neg g \rightarrow \neg g W r)$ be an LTL property where the protagonist controls $g$ and the antagonist controls $r$. An input-based game of $\varphi$ is shown in Figure 2.5(a). A turn-based game of $\varphi$ is shown in Figure 2.5(b). The turn-based game is equivalent to the input-based game in which the protagonist player moves first followed by the antagonist.

The turn-based game has been obtained from the input-based game as described in Chapter 6. Some states have labels that are a tuple of a state and input letter pair, these states represent the that one of the player has made its move (selected a letter from the respective alphabet). For instance the state labeled $(q_1, \neg g, r)(q_1, g, r)$ represents the state of the game where the play was in the state $q_1$ and the antagonist has set $r = \top$ after the protagonist set $g$ to either $\top$ or $\bot$.

As mentioned earlier, turn-based games are games of perfect information because each player has complete history of its own moves and the opponents moves. On the other hand, input-based games as defined above may give the impression that these are not games of perfect information (because the protagonist has to complete its move with the partial knowledge of the antagonist’s move). When the correspondence between the turn-based games and input-based games is established in Chapter 5 and 6, it becomes apparent that the transition from one state to another in an input-based game should not be considered an atomic operation. A single move of the protagonist and the antagonist in the input-based game corresponds to two or more moves of the protagonist and antagonist in the turn-based games. In other words, when the history of the game takes into account the Players’ choice of a letter from their respective alphabet, then input-based games are also seen as games of perfect information.

The existence and computation of winning strategies are central problems in the study of these games. A strategy is a function that defines the letter or successor of the current state a player should choose at each move. A strategy for Player $i$ in a turn-based game can be defined equivalently as either a function $\tau_i : Q^* \times Q_i \rightarrow Q_i$, or as a function $\tau_i : S_i \times Q_i \rightarrow S_i \times Q$. The set $S_i$ is Player $i$’s memory, which contains an initial element $\tilde{s}_i$. According to the cardinality of $S_i$, strategies are classified as infinite memory, finite memory, and memoryless (or positional). A strategy $\tau_i$ is winning for Player $i$ from a given state of the game iff victory is secured from that state regardless of the opponent’s choices as long as Player $i$ plays
according to $\tau_i$. In an input-based game, a strategy for Player 0 can be defined equivalently as either a pair of functions:

$$
\tau^1_0 : Q^* \times Q \rightarrow \Sigma_{ed} \quad \text{and} \quad \tau^2_0 : Q^* \times Q \times \Sigma_{ed} \times \Sigma_s \rightarrow \Sigma_{ep},
$$

or as a pair of functions:

$$
\tau^1_0 : S_0 \times Q \rightarrow \Sigma_{ed} \quad \text{and} \quad \tau^2_0 : S_0 \times Q \times \Sigma_{ed} \times \Sigma_s \rightarrow S_0 \times \Sigma_{ep}.
$$

A strategy for Player 1 can be defined equivalently as a function: $\tau_1 : Q^* \times Q \times \Sigma_{ed} \rightarrow \Sigma_s$ or as function: $\tau_1 : S_1 \times Q \times \Sigma_{ed} \rightarrow S_1 \times \Sigma_s$.

It is important to note that in this thesis, a set of $\omega$-regular properties is translated to a deterministic automaton $A = \langle \Sigma, Q, q_{in}, \delta, F \rangle$ with an input alphabet $\Sigma_{ed} \times \Sigma_s \times \Sigma_{ep}$, which is interpreted as an input-based game $G = \langle \Sigma, Q, D, q_{in}, \delta, F \rangle$ between the system (protagonist) and the environment (antagonist), where $D = Q \times (\Sigma_{ed} \times \Sigma_s \times \Sigma_{ep})$. From this game, a Mealy or Moore machine $M = \langle \Sigma_{ed} \times \Sigma_{ep}, \Sigma_s, Q, q_{in}, \delta, O \rangle$ is constructed, where transition function $\delta$ is derived from the input-based game $G$, while the output function is obtained from the winning strategy of the system in the input-based game $G$.

As mentioned in Section 2.1, the infinite set $F$ in the definition of either games can be effectively represented by an acceptance condition $\alpha$ defined in Section 2.3. The following two definitions are used in Chapter 4 which discusses the construction of turn-based games from the given specification. The definitions introduce the notation used in this thesis when a new game is obtained from another game by manipulating the acceptance condition $\alpha$ as discussed in Section 2.3.

**Definition 2.7.6.** For game $G = \langle \Sigma, Q, Q_0, Q_1, \delta, \alpha \rangle$, $G[\alpha \leftarrow \alpha']$ denotes the game $\langle \Sigma, Q, Q_0, Q_1, \delta, \alpha' \rangle$ where the domain of the acceptance condition $\alpha'$ is also the set of states $Q$.

**Definition 2.7.7.** Given game $G = \langle \Sigma, Q, Q_0, Q_1, \delta, \alpha \rangle$, the acceptance condition $\alpha'$ with the set of states $Q$ as its domain, is equivalent to $\alpha$ with respect to $G$ if a play is winning in $G$ iff it is winning in $G[\alpha \leftarrow \alpha']$. 
2.7.1 Specifications and Moore Machines

Given a specification and a set of input variables $X$, if the user partitions $X$ such that $X_{ed} = \emptyset$ then the specification is converted to a game between the system and the environment, where the system makes its move before the environment (this is discussed in detail in Section 5.12). If the system wins this game then one can easily generate a Moore implementation. If the system does not win this game then the specification may still be realizable. On the other hand, the user may partition $X$ such that $X_{ep} = \emptyset$ then the specification is converted to a game where the environment makes its move before the system. If the system wins this game, then generating a Moore implementation from system’s winning strategy in from the second game is not a trivial task (as it may not even exist).

In many cases, Moore implementations are more desirable in practice as discussed in Section 2.1. The following example shows the difference between the system’s winning strategy when the game is played with the objective of finding a Moore implementation and when the game is played in its full generality.

Example 2.7.8. Consider the property

$$\varphi = G(h \rightarrow g_0 \rightarrow X \neg m) \land (h \rightarrow g_1 \rightarrow X m) \land ((g_0 \land g_1) \leftrightarrow X p).$$

The property $\varphi$ is converted to an input-based game shown in Figure 2.6. If one plays the first type of game then system wins from $\{q_1\}$, otherwise system wins from $\{q_1, q_2\}$. When a Moore machine is generated which satisfies $\varphi$ then $\neg g_0 \lor \neg g_1$ must hold. Therefore by playing a game where the system plays first, one can focus only on finding those winning strategies which can be implemented by a Moore machine.

The above example illustrates the main reason why a user may want to partition the inputs $X_e$ controlled by the environment into two sets; $X_{ed}$ the set of disclosed inputs and $X_{ep}$ the set of private inputs.

2.8 Symbolic Representations and Variable Ordering

The design, implementation and verification of hardware based on a binary system manipulate large propositional formulae. The importance of efficient ways of representing and manipulating these formulae cannot be understated. Various representations such as And-Inverter Graphs (AIGs) [Hel63] and Binary
Decision Diagrams (BDDs) [Bry86] have been developed over the years. Each of these representations have strengths and weaknesses. Properly used, they can significantly improve the performance of an application. One focus of this thesis is to efficiently use BDDs for the automatic synthesis of a specification.

A Boolean function can be represented as a directed acyclic graph (DAG) with a single root, which consists of several decision nodes and leaf nodes. There are two types of leaf nodes called 0-leaf and 1-leaf. Each decision node is labeled by Boolean variable and has two child nodes called false child and true child. The edge from a node to a false (true) child represents an assignment of to 0 (1)\(^5\). Such a BDD is called ordered if different variables appear in the same order on all paths from the root. A BDD is said to be reduced if the following two rules have been applied to its graph:

- Merge any isomorphic subgraphs in the DAG of the Boolean function.
- No two nodes in the DAG have the same two children.

In popular usage, the term BDD almost always refers to Reduced Ordered Binary Decision Diagram (ROBDD) The advantage of an ROBDD is that it is canonical (unique) for a particular function and variable order [Bry86]. This property makes it useful in functional equivalence checking and checking the satisfiability of a Boolean formula. A path from the root node to the 1-leaf represents a (sometimes partial) variable assignment for which the represented Boolean function is satisfied. As the path descends to a low (high) child from a node, then that node’s variable is assigned to 0 (1). An example of a BDD is shown in Figure 2.8.

The following is a brief discussion on how symbolic representations are used for representing sets and computing various set operations.

**Definition 2.8.1.** Let \( S \) be a set and \( S' \) be a subset of \( S \). The characteristic function of \( S' \) is the mapping \( I_{S'} : S \to \{ \bot, \top \} \) defined by

\[
I_{S'}(x) = \begin{cases} 
\top & x \in S' \\
\bot & \text{otherwise} 
\end{cases}
\]

\(^5\) In the figures, the edge to false child is distinguished from an edge to a true child by a circle drawn on the edge.
A finite set $S$ can be represented symbolically using $n = \lceil \log_2 |S| \rceil$ binary variables. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of $n$ binary variables and let the set $A(X) \subseteq B(X)$ (discussed in Section 2.2) be the set of atomic Boolean functions (minterms); that is,

$$A(X) = \{-x_1 \land -x_2 \land \ldots \land -x_n, -x_1 \land -x_2 \land \ldots \land x_n, \ldots, x_1 \land x_2 \land \ldots \land x_n\} \quad (2.12)$$

and $|A(X)| = 2^n$. A mapping function (more commonly known as the encoding of $S$) $\Gamma : S \rightarrow A(X)$ is non-surjective when $|S| < 2^n$. The point-wise extension of the mapping function $\Gamma$ to a set of states $S$ is a Boolean function $\Gamma(S) \in B_+^\uparrow(A(X))$, such that

$$\Gamma(S) = \bigvee_{s \in S} \Gamma(s) \ .$$

The Boolean function representing the characteristic function of a subset $S \in S$ is given by $\Gamma(S)$. For sets $S$ and $A(X)$, the size of a BDD representation of a set $S \subseteq S$ depends in general on the choice of the mapping function $\Gamma$ as shown in the following example.

**Example 2.8.2.** Let $S = \{i, d, s\}$ and $X = \{x_0, x_1\}$. Then $A(X) = \{-x_1 \land -x_0, -x_1 \land x_0, x_1 \land -x_0, x_1 \land x_0\}$. Two mapping functions $\Gamma_1$ and $\Gamma_2$ are defined below:

$$\begin{align*}
\Gamma_1(i) &= -x_1 \land -x_0 & \Gamma_2(i) &= x_1 \land -x_0 \\
\Gamma_1(d) &= -x_1 \land x_0 & \Gamma_2(d) &= -x_1 \land x_0 \\
\Gamma_1(s) &= x_1 \land -x_0 & \Gamma_2(s) &= -x_1 \land -x_0 \ .
\end{align*}$$

Then the characteristic functions of $S = \{d, s\}$ for the two encodings are:

$$\begin{align*}
\Gamma_1(S) &= (\neg x_1 \land x_0) \lor (x_1 \land \neg x_0) \\
\Gamma_2(S) &= \neg x_1 \ .
\end{align*}$$

The BDDs for the two characteristic functions $\Gamma_1(S)$ and $\Gamma_2(S)$ are shown in Figure 2.7.

Apart from the mapping function, a good variable order often reduces the size of BDDs and therefore causes substantial gains in performance of BDD operations. This fact is illustrated in Fig. 2.8. There has been significant effort in obtaining a good variable order [Rud93].
Example 2.8.3. The sets $S$, $S$, $X$ and $A(X)$ are given in Example 2.8.2. A new mapping function $\Gamma$ is defined below:

$$\Gamma(i) = \{ \neg x_1 \land \neg x_0 \}$$
$$\Gamma(d) = \{ \neg x_1 \land x_0 \}$$
$$\Gamma(s) = \{ x_1 \land \neg x_0 , x_1 \land x_0 \} .$$

Then a characteristic function of $S$ for the mapping function $\Gamma$ is:

$$\Gamma(S) = x_1 \lor x_0 .$$

The BDDs for $\Gamma(S)$ is shown in Figure 2.7.

BDD’s are effectively used in applications that perform set operations on the subsets of the universe denoted by the set $S$. In such cases, it is extremely hard to select a good mapping function that will result in best representations of the subsets of $S$, when these subsets are not known in advance. As seen in Example 2.8.3, the size of the representation of a subset also depends on variable ordering, which further complicates the selection of a good mapping function. Therefore, a mapping function $\Gamma$ for $S$ is selected randomly.

However, when $\Gamma$ is not surjective then a more general mapping function $\Gamma : S \rightarrow B^+_\downarrow(A(X))$ can be selected, which often yields smaller symbolic representations of various subsets of $S$. A mapping function $\Gamma$ of this form is such that

$$\forall s_1 \in S . \forall s_2 \in S . s_1 \neq s_2 \rightarrow \Gamma(s_1) \land \Gamma(s_2) = \bot \quad \text{and} \quad \bigvee_{s \in S} \Gamma(s) = \top .$$

Let $\gamma(s) = \bigvee_{f \in \Gamma(s)} f$ be the disjunction of the elements in $\Gamma(s)$. The characteristic function of the set $S \subseteq S$ is given by a Boolean function $\Gamma(S)$ described as follows:

$$\Gamma(S) \leq \bigvee_{s \in S} \gamma(s) \quad \text{and} \quad \forall s \in S . \Gamma(S) \land \gamma(s) \neq \bot .$$

Mapping functions used in Vis $[B^+96]$ are of the more general type that have been described above, which often results in a significant overall performance gain.
The focus of this thesis is to perform operations on automata and games. Therefore the symbolic representations of automata and games are reviewed. Let \( A = \langle \Sigma, Q, q_{\text{in}}, \delta, \Pi \rangle \) be a disjunctive or a conjunctive parity automaton, where \( \Pi = \{ \pi_1, \pi_2, \ldots, \pi_m \} \). Let the encoding of the sets \( \Sigma \) and \( Q \) be given by the mapping functions \( \Gamma_{\Sigma} : \Sigma \rightarrow A(X) \) and \( \Gamma_Q : Q \rightarrow A(S) \), where \( X = \{ x_1, x_2, \ldots, x_n \} \), \( S = \{ s_1, s_2, \ldots, s_k \} \), \( n = \lceil \log_2 |\Sigma| \rceil \) and \( k = \lceil \log_2 |Q| \rceil \). One can obtain a symbolic representation of automaton \( A \) given by the tuple

\[
\hat{A} = \langle \Gamma_{\Sigma}, \Gamma_Q, I_{\text{in}}, \Delta(S, X, S'), \mathcal{P} \rangle.
\]

The predicate \( I_{\text{in}} = \Gamma(q_{\text{in}}) \). The transition function \( \delta(q, \sigma) \) is encoded as a transition relation \( \Delta(S, X, S') \) such that

\[
\Delta(\Gamma_Q(q), \Gamma_{\Sigma}(\sigma), \Gamma_Q'(q')) = \begin{cases} 
\top & q' \in \delta(q, \sigma) \\
\bot & \text{otherwise} 
\end{cases}.
\]

The set of parity conditions \( \Pi \) is represented by \( \mathcal{P} = \{ \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m \} \), where \( \mathcal{P}_i \) is a finite set of characteristic functions \( \{ \mathcal{P}_{i0}, \mathcal{P}_{i1}, \ldots, \mathcal{P}_{ik_i} \} \) such that \( k_i \) is the maximum parity with respect to \( \pi_i \) and

\[
\mathcal{P}_j(\Gamma_Q(q)) = \begin{cases} 
\top & \pi_i(q) = j \\
\bot & \text{otherwise}.
\end{cases}
\]

The symbolic representation of automata with different acceptance conditions follows the same principles. For instance, one characteristic function is sufficient to represent Büchi or co-Büchi acceptance condition. While Rabin and Streett acceptance conditions are represented by a set \( \mathcal{P} \), where each element is a pair of characteristic functions, one for the set \( U_i \) and the other for the set \( E_i \). A Muller acceptance \( \mathcal{M} \) condition can be represented by a set of characteristic functions where each characteristic function represents some set \( M \in \mathcal{M} \).

The symbolic representation of a turn-based game \( G = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle \) is given by the tuple

\[
\hat{G} = \langle \Gamma_{\Sigma}, \Gamma_Q, I_{Q_0}, I_{Q_1}, I_{\text{in}}, \Delta(S, X, S'), \mathcal{P} \rangle.
\]

\(^6\) \( S' = \{ s'_1, s'_2, \ldots, s'_n \} \) is the set of binary variables that encode the next state. The mapping function \( \Gamma_Q : Q \rightarrow A(S') \) denoted by \( \Gamma_Q' \) is obtained from \( \Gamma_Q \) such that for every \( q \in Q \) the Boolean function \( \Gamma_Q'(q) \) is the Boolean function \( \Gamma_Q(q) \) where every variable \( s_i \in S \) is replaced with the variable \( s'_i \in S' \).
The Boolean functions $I_{Q_0} = \Gamma(Q_0)$ and $I_{Q_1} = \Gamma(Q_1)$ represent the states that belong to Player 0 and Player 1 respectively. The Boolean functions $I_{Q_0}$ and $I_{Q_1}$ are complement of each other with respect to the Boolean function $\Gamma_Q(Q)$.

For the input-based game $G = \langle \Sigma, Q, D, \delta, \Pi \rangle$ where $\Sigma = \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep}$, the set of input variables $X$ is partitioned into the sets $X_{ed}$, $X_{ep}$ and $X_s$, where Player 0 controls $X_{ed}$ and $X_{ep}$ while Player 1 controls $X_s$. Instead of defining the Boolean functions $I_{Q_0}$ and $I_{Q_1}$, another Boolean function $I_D(S, X)$ is defined that represent the set $D$.

It is often the case that in automata not all states are reachable from the initial state. If the set of unreachable states is known then the encoding and the characteristic functions of the automata can be simplified further. In the context of synthesis from a set of $\omega$-regular properties, this aspect is further discussed in Section 5.1 in the context of games.

It has already been discussed that the size of a BDD is determined by the Boolean function being represented, the chosen ordering of the binary variables and the mapping function. In the worst case the size of a BDD is exponential in $|x|$ for any variable order of $x$. In some cases, given $|x|$ the size of a BDD ranges from linear to exponential for a particular variable order of $x$. Unless a good variable order is known in advance, applications using BDDs make an attempt to improve the variable order during the course of execution [Rud93].
Figure 2.5: (a) An input-based game obtained from $\varphi$. (b) A turn-based game obtained from $\varphi$. 
Figure 2.6: $G((h \rightarrow g_0 \rightarrow \mathbf{X} \neg m) \land (h \rightarrow g_1 \rightarrow \mathbf{X} m) \land ((g_0 \land g_1) \leftrightarrow \mathbf{X} p))$.

Figure 2.7: The BDDs for $\Gamma_2(S)$, $\Gamma_1(S)$ and $\Gamma(S)$ with variable order $x_1, x_0$.

Figure 2.8: The BDDs for the function $f$ with two different variable orders, $f = (a \land b) \lor (a \land c)$.
Chapter 3

Synthesis of Reactive Systems

From the time when the original approach of Pnueli and Rosner [PR89] was proposed, there have been significant advances in automatic synthesis of reactive systems. These improvements have renewed the hope that the industrial adoption of automatic synthesis approaches will happen in the near future. This chapter provides the formal description of the game-theoretic approach to synthesis and gives an overview of various game-theoretic algorithms that have been developed. Wherever it is appropriate the virtues and weaknesses of various techniques are discussed. The approach presented in this thesis builds upon some of this foundational work, which is therefore discussed in more detail.

3.1 Satisfiability vs. Realizability

Consider an $\omega$-regular property $\varphi$ over the alphabet $\Sigma$. The satisfiability problem for $\varphi$ is to decide whether there exists a word $w = \sigma^0, \sigma^1, \sigma^2, \ldots$ that satisfies $\varphi$. In other words, $\varphi$ is satisfiable if and only if the language described by $\varphi$ is not empty. Checking the satisfiability of an $\omega$-regular property is encountered in LTL model checking. Given a model $M$ and an LTL property $\varphi$, which describes the intended behavior of the model, the problem of verifying that $M$ always behaves according to $\varphi$ is reduced to checking if $M$ satisfies the property $\neg \varphi$. If the answer is yes then the model does not behave according to $\varphi$, otherwise it is proven that $M$ behaves according to $\varphi$.

To check the realizability of $\varphi$, one has to suppose that $\Sigma = \Sigma_e \times \Sigma_s$ because the property is defined over the inputs and outputs. Then the realizability problem for $\varphi$ is to decide whether a model $M$ exists with an input alphabet $\Sigma_e$ and an output alphabet $\Sigma_s$ that guarantees the satisfaction of $\varphi$. Therefore, the
realizability problem of $\varphi$ is to decide whether for every input letter there exists an output letter such that the input-output infinite word satisfies $\varphi$. The notion of realizability of $\varphi$ is directly related to two-player games where the input player is the antagonist and the output player is the protagonist. The satisfiability of $\varphi$ is related to verification and the realizability of $\varphi$ is related to synthesis.

Therefore, in the game obtained from $\varphi$, one is interested in knowing if there exists a strategy $\tau : \Sigma_e^* \rightarrow \Sigma_s$ for the output player such that for every input word $\sigma_0^0, \sigma_1^1, \sigma_2^2 \ldots \in \Sigma_e^\omega$ the input-output word

$$w = (\sigma_0^0, \tau(\sigma_0^0)), (\sigma_1^1, \tau(\sigma_1^1)), (\sigma_2^2, \tau(\sigma_2^1)), \ldots$$

satisfies $\varphi$. If $\tau$ exists then the property is realizable. Since the property $\varphi$ describes an $\omega$-regular language then the strategy function $\tau$ only uses finite memory [Tho95]. Therefore, the strategy function $\tau$ can be implemented as a transducer (described in Section 2.1) with input language $\Sigma_e^\omega$ and an output language which is a subset of $\Sigma_s^\omega$. The implementation of $\tau$ is the model $M$ which behaves according to $\varphi$.

As briefly discussed in Chapter 2, the transducer $M$ is generally a Mealy model, but may be reduced to a Moore model such that for every input word $\sigma_0^0, \sigma_1^1, \sigma_2^2 \ldots \in \Sigma_e^\omega$ the input-output word

$$w = (\sigma_0^0, \tau(\epsilon)), (\sigma_1^1, \tau(\sigma_0^0)), (\sigma_2^2, \tau(\sigma_1^1)), \ldots$$

satisfies $\varphi$ (where $\epsilon$ is the empty string). Notice that in a Moore model, the choice of the $i$-th output letter is not affected by the $i$-th input letter.

### 3.2 Game Based Approaches

The synthesis of reactive systems from $\omega$-regular specifications was originally proposed in [PR89]. It was proven that if the specification describes a centralized\(^1\) reactive system then the problem is decidable. When the specification of a reactive system with inputs $X_e$ and outputs $X_s$ was given by a set of LTL formulas over the signals $X_e \cup X_s$, the specification was translated to an NBW and then determinized to DRW using Safra’s [Saf88] determinization procedure. The need for determinization and the details of the process are provided in Section 3.5 and 3.6. The DRW was then interpreted as a game between the system

\(^1\) In [PR90], Pnueli and Rosner proved that synthesis of a distributed reactive system in its full generality is undecidable, while it has non-elementary complexity for pipelined architectures.
and the environment, where a win for the system meant that the specification was realizable, while a win for
the environment meant that the specification was unrealizable. It was shown that an implementation for the
specification could be obtained from the winning strategy of the system in the synthesis game.

This approach suffers from a doubly exponential bound because the conversion of an LTL formula
to an automaton suffers from an exponential blowup, while the determinization of the automaton also suf-
fers from an exponential blowup. Time complexity of the whole synthesis procedure and the size of the
implementation, both equal to $O(2^{2^n})$, for some constant $c$, where $n$ is the size of the LTL formula $\varphi$.

### 3.3 Synthesis from General Reactive (1) Specifications

A specification belongs to the GR(1) class if it is in the following form:

$$\bigwedge_{1 \leq i \leq p} (GF K_i) \rightarrow \bigwedge_{1 \leq i \leq n} (GF J_i), \quad (3.1)$$

where $J_i$ and $K_i$ are past-time LTL formulae. This specification can be converted to a game with a gen-
eralized Streett(1) acceptance condition defined in Section 2.4. A specialized synthesis approach for specifi-
cations belonging to the GR(1) class was proposed in [PPS06]. This approach is briefly discussed in this
section.

Given a turn-based game $G = \langle \Sigma, Q, Q_0, Q_1, \delta_{det}, \Phi \rangle$, where $\Phi$ describes an acceptance condition of
the form of (3.1), one can solve this game through a $\mu$-calculus formula described in (3.2). This formula
solves the game in time $O(n^3)$, where $n$ is the number of recurrence formulae in the consequent of (3.1). For
the sake of simplicity, let $J_i$ and $K_i$ be propositional formulae such that $J_i \in B(Q)$ and $K_i \in B(Q)$. (When
either $J_i$ or $K_i$ is a past-time LTL formula, then it is converted to a deterministic Büchi automaton, which is
composed with $G$. The acceptance condition of the Büchi automaton is expressed as a propositional formula
over the labels of the states of the automaton.) The winning states of the game $G$ are the states from which
the protagonist can force a play that satisfies the acceptance condition $\Phi$. These states can be computed by
the following $\mu$-calculus formula:

$$W = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} \left[ \mu Y (\nu X (J_1 \land \bigotimes Z_2 \lor \bigotimes Y \lor \bigvee_{1 \leq i \leq p} -K_i \land \bigotimes X)) \right]$$

This formula describes the restricted pre-image operator in the turn-based game $G$. A state $q$ belonging to the antagonist (that is, in $Q_0$) is only included in the pre-image of $S$ iff from $q$ the antagonist has no choice but to visit a state in $S$. A state $q$ belonging to the protagonist is included in the pre-image of $S$ iff from $q$ the protagonist has at least one way to visit some state in $S$.

The matrix notation describes a formula which may be viewed as the mutual fixpoint of the variables $Z_1, Z_2, \ldots, Z_n$. It is equivalent to a formula where a single variable $Z$ replaces the variables $Z_1, Z_2, \ldots, Z_n$, and the variable $Z$ ranges over pairs a $n$-tuple of states. Intuitively the $\mu$-calculus formula is computing the greatest fixpoint $Z_i$ in the $i$th row. This fixpoint is the set of states that can eventually satisfy $J_i$ or continuously falsify some $K_i$. These set of states must also be able to visit in one step (if these states can reach the states computed in the next row in multiple steps then they must also be able to reach them in one step) the states computed in the next row.

A game with an acceptance condition described by the LTL formula in the form of (3.1) can be equipped with a generalized parity condition $\Pi = \{-\Pi_K, \Pi_J\}$ (defined in Section 2.3), where $\Pi_K = \{\hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_p\}$ and $\Pi_J = \{\pi_1, \pi_2, \ldots, \pi_n\}$ are conjunctive parity conditions. ($-\Pi_K = \{-\hat{\pi}_1, -\hat{\pi}_2, \ldots, -\hat{\pi}_p\}$ is a disjunctive parity condition and the generalized parity condition $\Pi$ can be written as $\{-\hat{\pi}_i\}, \{-\hat{\pi}_2\}, \ldots, \{-\hat{\pi}_p\}, \Pi\}$. In this game, the odd player wins by either winning one of the conditions $\{-\hat{\pi}_i\}$
or the conjunctive parity condition $\Pi$.) The parity condition $\hat{\pi}_i \in \Pi_K$ ($\neg \hat{\pi}_i \in \neg \Pi_K$) assigns the priority 1 (2) to states of $G$ that satisfy the propositional formula $K_i$ and all the other states are assigned priority 0 (1). Similarly, the priority condition $\pi_i \in \Pi_J$ assigns the priority 1 to the states of $G$ that satisfy the propositional formula $J_i$ and all the other states are assigned priority 0. As one shall realize later in this chapter, an algorithm to solve a generalized parity condition does not exist yet. If one was forced to solve this game through a parity game solver then one could convert this game to a non-generalized parity condition as discussed in Section 2.4. For these type of games, an efficient procedure to solve a game $G$ with the generalized parity acceptance condition $\Pi$ obtained from (3.1) is discussed in Section 4.3.1 (the game $G$ is not converted to a game with nongeneralized parity condition rather a unique property of (3.1) is exploited).

### 3.4 LTL to Automaton

One of the common model checking problems is to check if a model $M$ satisfies the LTL formula $\varphi$. One way to check this is by obtaining an $\omega$-automaton $A$ for the negation of the LTL formula $\varphi$. Model and automaton are then composed to obtain $M \times A$. If $M \times A$ accepts the empty language then the model $M$ satisfies the LTL formula $\varphi$, otherwise the model fails the formula $\varphi$. In the game-theoretic approach to synthesis of a reactive system, the LTL formula $\varphi$ that describes the system’s behavior is translated to an automaton $A$; a turn-based game $G$ between the system and the environment is then obtained from $A$. If the system has a winning strategy in $G$ then the formula $\varphi$ is realizable.

The conversion of the LTL formula to an automaton is briefly discussed here. It has been discussed in Section 2.6 that an LTL formula $\varphi$ describes an $\omega$-regular language $L(\varphi)$, hence there exists an $\omega$-automaton that accepts the language $L(\varphi)$. In model checking, the formula $\neg \varphi$ is converted to an NBW $A$, because the Büchi acceptance condition is simple, and nondeterminism does not impede the check for language emptiness of the composition of $A$ with the given model. The translation of LTL formulae to $\omega$-automata with applications in automata based LTL model checking has been extensively studied [VW94, GPVW95, SB00, GO01, Sch01, Thi02, ST03].

The translation of an LTL formula to an $\omega$-automata often results in a generalized NBW, that is, in an NBW with a set of Büchi acceptance conditions. Although the de-generalization of NBW has been
described in Section 2.4, it is not required for the purpose of LTL model checking. Model checkers can check the language emptiness of a generalized Büchi automaton. The translation of an LTL formula into an automaton with a set of Büchi acceptance conditions relies on the following expansion rules.

- $\psi \cup \varphi$ iff $\varphi \lor (\psi \land X \psi \cup \varphi)$
- $\psi \cap \varphi$ iff $\varphi \land (\psi \lor X (\psi \cap \varphi))$

Given an LTL formula $\varphi$, an initial state $q_{in}$ that represents $\varphi$ is created. A new state is created for any sub-formula of $\varphi$ that is encountered during the following procedure. In the next step, the expansion rules are applied until a formula $\Phi$ is obtained, which is equivalent to $\varphi$, that consists of only formulae beginning with an $X$, propositional formulae and Boolean connectives. This formula $\Phi$ is then rewritten into disjunctive normal form. The formula $\varphi$ has now been rewritten in such a way that in every disjunct there is a propositional part that has to be true now, and a part that has to be true in the next state. A new state $q$ is created for any formulae preceded by a $X$ operator that do not have a state associated with them yet. For every disjunct of $\varphi$, for instance $p \land q \land X \varphi$, an arc is added labeled $p \land q$ from the state corresponding to $\varphi$ to the state corresponding to $\psi$. The process is repeated for the sub-formula $\psi$.

The transitions from one state to another need to be distinguished into those that satisfy acceptance conditions and those that do not. An acceptance condition is created for each $\textbf{until}$ operator in $\varphi$. A transition between states satisfying the $p \cup q$ formula, but not satisfying $q$ is not accepting, while all the others are. Once transitions are labeled with the acceptance conditions, the acceptance condition can be moved to the states. If all transitions into a state have identical acceptance conditions, then the acceptance conditions are directly moved to the state. Otherwise, the state needs to be split such that every resulting copy has incoming transitions with identical acceptance conditions.
Consider the LTL formula $\varphi = G(p \rightarrow X F q)$. Let $\psi = F q$ then:

$$
\varphi = G(p \rightarrow X F q) \\
= (\neg p \lor X F q) \land X \varphi \\
= (\neg p \land X \varphi) \lor X(\psi \land \varphi)
$$

$$
\psi \land \varphi = (q \lor X F q) \land [(\neg p \land X \varphi) \lor X(\psi \land \varphi)]
\\
= (\neg p \land q \land X \varphi) \lor (q \land X(\psi \land \varphi)) \lor (\neg p \land X(\psi \land \varphi)) \lor X(\psi \land \varphi)
$$

The second and third terms of the expansion of $\psi \land \varphi$ are subsumed by the fourth and last term. However, the second term is an accepting transition, while the fourth term is not. Therefore, only the third term can be dropped. The resulting automaton is shown on the left in Figure 3.1. The accepting transitions are marked with a black dot. These show that the top state is accepting, while the bottom state should be split. The result is shown on the right in Figure 3.1.

The translation of an LTL formula to a Büchi automaton may result in an exponential blowup, but many LTL formulae can be represented by reasonably small NBWs. The interested reader can refer to the cited literature.
3.5 Determinization and Games

In the original game-based approach to synthesis briefly described in Section 3.2, the specification is translated to a deterministic Rabin game. The importance of determinism in games can be understood through the concept of simulation [Mil71]. Given two Büchi automata $A = \langle \Sigma, Q_A, a_{in}, \delta_A, B_A \rangle$ and $B = \langle \Sigma, Q_B, b_{in}, \delta_B, B_B \rangle$ such that $L(A) = L(B)$, then that does not necessarily mean that $A$ can simulate $B$. This fact is illustrated in Figure 3.2. This observation is essential to understanding why determinization is important in infinite games with $\omega$-regular winning conditions. Various types of simulation [HKR97] between two automata are defined below.

**Definition 3.5.1.** Automaton $A$ simulates automaton $B$, written $B \preceq A$, if there exists a simulation relation $\mathcal{R} \subseteq Q_A \times Q_B$ such that $(a_{in}, b_{in}) \in \mathcal{R}$ and for every $(a, b) \in \mathcal{R}$ and for all the letters $\sigma \in \Sigma$ and for all states $a' \in \delta_B(a, \sigma)$ there exists a state $b' \in \delta_B(b, \sigma)$ such that $(a', b') \in \mathcal{R}$.

**Definition 3.5.2.** Automaton $A$ directly simulates automaton $B$, written $B \preceq_d A$, if $B \preceq A$ and for every word $w \in \Sigma^\omega$ the runs $a_{in}, a_1, a_2 \ldots$ and $b_{in}, b_1, b_2, \ldots$ are such that $a_{in} \in B_A \iff b_{in} \in B_B$ and $\forall i > 0. a_i \in B_A \iff b_i \in B_B$.

**Definition 3.5.3.** Automaton $A$ fairly simulates automaton $B$, written $B \preceq_f A$, if $B \preceq A$ and for every word $w \in \Sigma^\omega$ the runs $\rho_A$ and $\rho_B$ are such that if $\rho_B$ is an accepting run then $\rho_A$ is also an accepting run.

The above definitions can be adapted for automata with other types of acceptance conditions described in Section 2.3. From the above definitions, one can notice that $B \preceq_d A$ implies $B \preceq_f A$ which implies $B \preceq A$. Furthermore, $B \preceq_f A$ implies that $L(B) \subseteq L(A)$. The implications in the other direction are not always true as can be seen in Example 3.5.4.

When confronted with the question whether $A$ fairly simulates $B$, one can answer this question by solving a fair simulation game between two players. In this game, in each move the antagonist selects the input letter $\sigma$ read by both automata and selects the next state of $B$ among those reachable via transitions enabled by $\sigma$, while the protagonist only selects the next state of $A$. If the protagonist has a winning strategy in this game then $A$ fairly simulates $B$; otherwise $A$ does not fairly simulate $B$. The fair simulation game is
defined as follows: Initially, the antagonist’s token is placed on $b_{in}$ and the protagonist’s token is placed on $a_{in}$. At each turn, let $a \in Q_A$ be the state with the protagonist’s token and let $b \in Q_B$ be the state with the antagonist’s token. Antagonist chooses a letter $\sigma \in \Sigma$ and moves the $B$ token to one of the states in $\delta_B(b, \sigma)$. Protagonist then moves the $A$ token to one of the states in $\delta_A(a, \sigma)$. Protagonist wins if run of $B$ is not an accepting run or the run of $A$ is an accepting run. A winning strategy for Player 1 is a function $\tau : (Q_A \times Q_B \times \Sigma)^+ \rightarrow Q_A$ that is consistent with $\delta_A$ (i.e., $\forall b \in Q_B . \forall a \in Q_A . \forall \sigma \in \Sigma . \tau(a, b, \sigma) \in \delta_A(a, \sigma)$) and that guarantees victory regardless of the opponent’s choices. Notice that the winning condition of a fair simulation game is given by an $\omega$-regular acceptance condition which can be described by the LTL formula $\mathsf{G} \mathsf{F} B_B \rightarrow \mathsf{G} \mathsf{F} B_A$.

The following example describes the case of two automata which are language equivalent but one cannot direct or even fair simulate the other.

**Example 3.5.4.** The two automata shown in Figure 3.2 (adapted from [KPP03]) accept the language $bc^\omega \lor bd^\omega$, where $A$ is nondeterministic and $B$ is deterministic. The fair simulation game to check whether $B \preceq_f A$ is lost by the protagonist. Hence, $A$ cannot simulate $B$. From the initial state, the automaton $A$ has to commit to only being able to continue its run for suffix $d^\omega$ or $c^\omega$ after reading the letter $b$, while the automaton $B$ has no such restriction in its initial state. (In this particular case, one observes that $B \not\preceq A$, though $B \preceq A$ is a pre-requisite for $A$ to be able to fair simulate $B$.)

On the other hand, the protagonist wins the fair simulation game to check if $A \preceq_f B$. (Notice that $A \preceq_d B$.)

It has already been discussed that checking the realizability of an $\omega$-regular property can be done by converting the $\omega$-regular property to a game. This game also has an $\omega$-regular winning condition such that a play in the game is winning iff it satisfies the winning condition. As it was the case in the fair simulation game, the antagonist may be able to use nondeterminism in the game against the protagonist. Example 3.5.4 shows that for the nondeterministic automaton $A$ and deterministic automaton $B$ one cannot check if $L(B) \subseteq L(A)$ by playing a fair simulation game. Similarly, the Example 3.5.5 shows that synthesis of $\varphi$ cannot be checked by obtaining a nondeterministic game from $\varphi$. 


Section 3.1 describes the process through which the realizability of an \( \omega \)-regular property is checked. In short, a deterministic automaton \( A_\varphi \) is obtained from \( \varphi \), then a game between the environment and the system is derived from \( A_\varphi \). In many cases, given a game structure\(^2\) and \( \varphi \), one is interested in checking the realizability of \( \varphi \) with respect to the game structure. In such cases, the automaton \( A_\varphi \) obtained from \( \varphi \) is composed with the given game structure. A game is obtained from the composition of game structure and \( A_\varphi \). The winning condition of the game is given by the winning condition of \( A_\varphi \). The following example illustrates the fact that the realizability of \( \varphi \) for the game structure \( G \) cannot be correctly computed through the nondeterministic automaton \( A_\varphi \) which accepts the language described by \( \varphi \).

**Example 3.5.5.** Consider the input-based game \( G \) shown in Figure 3.3. The protagonist controls \( q \), while the antagonist controls \( p \). In this game at the start of every play or whenever \( q \) is de-asserted after being asserted, the protagonist has to make sure that \( p \) and \( q \) are mutually exclusive until \( p \) is de-asserted. Given the formula \( \varphi = F G p \lor F G q \), the protagonist has a strategy such that every play of \( G \) satisfies \( \varphi \).

The simplest strategy for the protagonist is to assert \( q \) forever when \( p \) is de-asserted. If the antagonist never de-asserts \( p \) then the protagonist’s only option of de-asserting \( q \) produces a play that still satisfies \( \varphi \).

The property \( \varphi \) is accepted by the nondeterministic Büchi automaton \( A_\varphi \) shown on the right in Figure 3.3. Let the nondeterministic input-based game \( \mathcal{G} \) be the game obtained from the composition of \( G \) and

\(^2\) In this thesis, a game with no winning condition is termed as a game structure. In other words, a game structure is a game where every play of the game is a winning play.
Figure 3.3: Every run of $G$ is an accepting run. The nondeterministic Büchi automaton $A_\varphi$ accepts the language defined by the LTL formula $F G p \lor F G q$. The nondeterministic game $G$ is the composition of $G$ and $A_\varphi$.

In this game the antagonist controls $p$, while the protagonist controls $q$ and selects the next state of $A_\varphi$.

It can be observed that $G$ is deterministic but $G$ is not.

The protagonist does not have a winning strategy in $G$. If the antagonist insists on asserting $p$ forever, then the protagonist cannot choose the terminal state of $A_\varphi$, where the run can only continue if the antagonist cooperates and asserts $p$. The protagonist has to wait for the antagonist to de-assert $p$, but again the antagonist is not obliged to cooperate. Therefore, the antagonist can continue to assert $p$ forever, and the protagonist can only respond by de-asserting $q$ and selecting the initial state of $A_\varphi$ as the next state. This play is not a winning play of $G$. Therefore, there is no winning strategy for the protagonist in $G$.

The nondeterminism of $A_\varphi$ is used against the protagonist by the antagonist, which does not happen when $A_\varphi$ is deterministic. The Example 3.5.5 is continued below.

Example 3.5.6. The property $\varphi$ is converted to a deterministic co-Büchi automaton $A_\varphi$ shown on the right in Figure 3.4. Let $G$ be the deterministic input-based game obtained from the composition of $G$ and $A_\varphi$. In this game the antagonist controls $p$ and the protagonist controls $q$.

One can easily observe that the antagonist’s strategy of playing $p$ forever still produces a play which satisfies $\varphi$. The restriction of $G$ to protagonist’s winning strategy is denoted by $G_p$ shown on the middle of Figure 3.4.

One may wonder if determinization of the $\omega$-regular property $\varphi$ is always necessary. This issue is
Figure 3.4: Every run of $G$ is an accepting run. The deterministic co-Büchi automaton $A_\varphi$ accepts the language defined by the LTL formula $F G p \lor F G q$. The initial state of $A_\varphi$ is the non-accepting state. The deterministic game $G$ is the composition of $G$ with $A_\varphi$.

discussed in the next section where it is shown that the property $\varphi$ can be translated to specific types of nondeterministic automata that can be used to check the realizability of $\varphi$.

### 3.6 To Determinize or Not to Determinize?

Since determinization is expensive, there have been attempts to circumvent it. Section 3.10 discusses a completely different game-based approach to synthesis, which avoids the determinization step. In [HP06] it was shown that a nondeterministic automaton can be used to check the realizability of an $\omega$-regular property if the automaton is good for games (GFG). A nondeterministic $\omega$-automaton is good for games when it can fair simulate its deterministic counterpart. In such circumstances, the antagonist cannot use the nondeterminism of the automaton against the protagonist. Example 3.5.6 is continued below.

**Example 3.6.1.** The automaton $A_\varphi^n$ shown in the middle in Figure 3.5 for property $\varphi$ fair simulates the deterministic automaton $A_\varphi^d$ for $\varphi$ shown on the right. One can observe that the game obtained from composing $G$ shown on the left with $A_\varphi^n$ produces a game where the protagonist has a winning strategy.

One can often take advantage of a GFG automaton that is smaller than its deterministic counterpart. A procedure to obtain a GFG parity automaton from an NBW that is more efficient than the determinization procedure of an NBW (discussed in Section 3.7) was introduced in [HP06]. However, this procedure does not guarantee that the GFG automaton will be smaller than its deterministic counterpart in terms of the number of states. Furthermore, the complexity of the acceptance condition of the GFG automaton is at least
identical to the Rabin index (later defined in Definition 3.7.2) of the language. In other words, when NBW is converted to GFG parity automaton and a DPW then the number of colors in GFG parity automaton is at least equal to the number of colors of DPW. In fact, it is proven in Theorem 3.6.2 that the number of colors in GFG parity automaton cannot be less than the DPW that accepts the same language and has the fewest number of colors. In other words, the theorem states that a DPW of (minimum) index \( k \), cannot be fair simulated by any NPW with index less than \( k \).

**Theorem 3.6.2.** Let \( L \subseteq \Sigma^\omega \) be an \( \omega \)-regular language. Let \( N = (\Sigma, Q_N, \delta_N, \pi_N) \) be an NPW accepting \( L \) with \( \pi_N : Q_N \to [k] \) and \( D = (\Sigma, Q_D, \delta_D, \pi_D) \) be a DPW accepting \( L \) with \( \pi_D : Q_D \to [k'] \). Let \( k' \) be the minimum number of colors for \( L \) and \( k < k' \). Then \( N \) does not fair simulate \( D \).

**Proof.** Suppose a fair simulation strategy \( \tau : Q_N \times Q_D \times \Sigma \to Q_N \) exists for \( N \). (without loss of generality \( \tau \) is positional because the winning condition is the disjunction of two parity conditions.) Consider the DPW \( C = (\Sigma, Q, \delta, \pi) \), where

\[
Q = Q_N \times Q_D
\]

\[
\delta((q_N, q_D), \sigma) = (\tau(q_N, q_D, \sigma), \delta_D(q_D, \sigma))
\]

\[
\pi((q_N, q_D)) = \pi_N(q_N)
\]

Suppose \( w \in \Sigma^\omega \) is accepted by \( C \). Let \( \rho \) be the play of \( N \) on the word \( w \) when obeying the strategy \( \tau \). The play \( \rho \) is the projection on \( Q_N \) of the play of \( C \), moreover it is accepting because \( C \) accepts. Therefore, \( w \in L \). Conversely, if \( w \in L \), its play in \( D \) is accepting. Since \( \tau \) is a simulation strategy for \( N \), the play of
Nonw according to $\tau$ is also accepting. Hence the play of $C$ on $w$ is also accepting and $C$ is a DPW that accepts $L$ such that $\pi : Q \rightarrow [k]$. This, however, contradicts the assumption that $k' > k$ is minimum for $L$. Hence $\tau$ does not exits.

Thus the time savings coming from the use of GFG automata are due to simplification of the transition structure of the GFG automata. In other words, solving a game obtained from a GFG automaton is simpler iff its structure is simpler than its deterministic counterpart with minimum index. (The Example 3.6.1 shows a co-Büchi GFG automaton which is smaller than a DCW for the same language.) However, the procedure of [HP06] does not guarantee that a GFG automaton will be smaller than its deterministic counterpart. The size of the automaton is of extreme importance and in Section 5.1 it is argued that the game structure of the synthesis game is embedded in the implementation, which is obtained from the winning strategy of the protagonist. Therefore, in practice the deterministic parity automata are preferred over GFG parity automata as there is no procedure that can always generate a GFG automaton which is smaller than its deterministic counterpart.

When the specification is of the GR(1) type, then this implies that the properties in the specification can be decomposed into several recurrence properties. In practice, such a specification is presented as an implication between two sets of recurrence properties where each set is conjoined to form a single recurrence property. In most cases a simple deterministic automaton can be constructed for these properties without the need for determinization through the general determinization approach discussed in Section 3.7. Finally, the game obtained from a GR(1) specification can also be converted to a nongeneralized parity game or general Street(1) game as discussed in Section 2.4. Therefore, one does not need to worry about the cost of determinization.

In [HRS05] a simple construction of a GFG automaton for some LTL properties is discussed. A shift automaton, which offers the protagonist the ability to rectify a nondeterministic choice of the NBW, is composed with the NBW, the resulting automaton is a GFG automaton. Unfortunately, this approach is not complete and there exist LTL formulae for which no shift automaton exists. The LTL formulae that describe the languages in general reactivity class (defined in Section 2.6) are examples for which no shift automata
exist. However, in practice specifications that may benefit from this approach are not common.

In Section 3.10, another game-based approach checking the realizability of $\omega$-regular specification is discussed. This approach avoids the determinization step.

When synthesizing a transducer from an $\omega$-regular specification then one can either use deterministic games or use alternatives that do not require computationally expensive determinization step. However, avoiding determinization often comes at a price. In practice these alternative techniques have not proven to be a viable option because the efficiency and quality of the synthesis procedure suffers significantly.

3.7 Determinization of an NBW

In this section, the determinization of an NBW is discussed briefly. It was mentioned in Section 2.6 that DBWs are less expressive than NBWs. Therefore, there exist NBWs that can only be determinized to automata with more powerful acceptance conditions like DMWs, DPWs, DRWs, or DSWs. A nondeterministic finite automaton that accepts a regular language on finite words can be determinized through subset construction [Sco59]. However, determinization of a nondeterministic $\omega$-automaton cannot be done through the subset construction as described in Example 3.7.1. Fortunately, when it can be detected that the LTL formula is a syntactic safety property then the determinization procedure through the subset construction [Sco59] is sufficient.

Example 3.7.1. Consider the alphabet $\Sigma = \{-p, p\}$ and the NBW $A$ shown in Figure 3.6. The automaton $A$ accepts the language $L(A) = \Sigma^* p^\omega$ (described by the LTL formula $F G p$). When the subset construction procedure is applied to $A$, the resulting automaton is $D$ shown in Figure 3.6. One can derive four different DBWs from $D$ with four different Büchi acceptance conditions described below:

- $B = \emptyset$ then $D$ accepts the empty language
- $B = \{\{a\}\}$ then $D$ accepts the language defined by the LTL formula $G F \neg p$
- $B = \{\{a, b\}\}$ then $D$ accepts the language defined by the LTL formula $G F p$
None of these conditions are correct and hence subset construction cannot produce the desired DBW. One must equip the automaton $D$ with a co-Büchi acceptance condition $C = \{a\}$, then $L(D) = L(A)$.

However, DCW is also not powerful enough to represent all the $\omega$-regular languages. For instance, there are languages that are not covered by the union of DCW and DBW. Hence one is forced to use a more powerful condition like the Rabin acceptance condition.

The determinization of an NBW to a DRW was first proposed by McNaughton in [McN66]. This procedure suffered from a doubly exponential blow-up in the size of the NBW. In [Saf88], Safra showed that an NBW can converted to a DRW with only an exponential blow-up in the size of NBW. Given an NBW with $n$ states then one can generate a DRW with $2^{O(n \log n)}$ states and a Rabin condition with $n$ accepting pairs. The determinization of a NBW to a DRW was the main reasons why DRWs were used in the original synthesis approach of [PR89]. Safra’s determinization procedure and its variants keep track of all the possible runs of the NBW for an input word by building finite trees. The nodes of these trees, popularly known as Safra’s trees, are labeled with states of the NBW. After reading a letter from the input word, the deterministic automaton transits into another Safra tree. A unique Safra tree represents a state in the deterministic counterpart of the NBW. The deterministic automaton is then equipped with appropriate acceptance condition. Thus, a deterministic automaton for NBW is generated. A significantly improved version of Safra’s determinization procedure was proposed by Piterman in [Pit06]; it converts an NBW to a (nongeneralized) DPW.

When checking the realizability of an $\omega$-regular property that describes the language $L$, the complex-
ity of solving the synthesis game is in terms of the complexity of the deterministic automaton from which
the synthesis game was derived. The complexity of the automaton is determined by the size of the graph of
the automaton and by the acceptance condition. The complexity of the acceptance condition is directly gov-
erned by the language accepted by the deterministic ω-automaton. It was briefly mentioned in Section 2.6,
that ω-regular languages form an infinite hierarchy [Wag79]. Given an ω-regular language L, the position
of L in this hierarchy is based on the Rabin index of the language. The notion of Rabin index of L denoted
by $R_I(L)$ is defined below.

**Definition 3.7.2.** Given an ω-regular language L, the Rabin index is the minimum number of Rabin pairs
required to accept L by a DRW.

$$R_I(L) = \min\{|\mathcal{R}| \mid \exists A = \langle \Sigma, Q, q_{in}, \delta_{det}, \mathcal{R} \rangle \text{ such that } L = L(A) \} ,$$

where

$$\mathcal{R} = \{(U_1, E_1), (U_2, E_2), \ldots, (U_n, E_n)\} .$$

The Rabin index of an ω-regular language L can be computed from a deterministic ω-automaton that
accepts L. Let $A = \langle \Sigma, Q, q_{in}, \delta_{det}, \alpha \rangle$ be an ω-automaton that accepts the language L. A subset of Q is
effective if it equals inf($\rho$) for some run $\rho$ of $A$. A positive chain of length $m$ is a sequence of $m$ effective
sets $Q_1 \subset Q_2 \subset \cdots \subset Q_m$ such that $Q_1$ satisfies the acceptance condition of $A$ iff $i$ is odd. The Rabin index
$R_I(L)$ of language L is related to the maximal length $\Xi(A)$ of a positive chain in $A$ by

$$R_I(L) = \lceil (\Xi(A) + 1)/2 \rceil .$$

Carton and Maceiras have devised an algorithm that finds $R_I(L)$ given a DPW that recognizes L in
time $O(|Q|^2|\Sigma|)$ [CM99]. Moreover, every DPW P that recognizes L can be equipped with a new parity
condition $\pi : Q \rightarrow [\Xi(A) + 1]$ without changing the accepted language. The following procedure there-
fore yields a DBW from an NBW if one exists: Convert NBW $B$ to an equivalent DPW P by Piterman’s
procedure. Compute $\Xi(P)$ with the algorithm of Carton and Maceiras. If $\Xi(P) \leq 1$ the equivalent par-
ity condition with $\leq 2$ priorities computed together with $\Xi(D)$ can be interpreted as a Büchi acceptance
condition; otherwise no DBW equivalent to $B$ exists. (If $\Xi(P) = 0$, $B$ accepts the empty language.)
Given NBW $B = \langle \Sigma, Q_B, b_{in}, \delta_B, \mathcal{B} \rangle$ which accepts the $\omega$-regular language $\mathcal{L}$, determination of $B$ to DPW $P = \langle \Sigma, Q_P, p_{in}, \delta_P, \pi \rangle$ through Piterman’s procedure offers several advantages over the determination of $B$ to a DRW $R = \langle \Sigma, Q_R, r_{in}, \delta_R, \mathcal{R} \rangle$ through Safra’s procedure:

1. $P$ has fewer states than $R$ and when $P$ is interpreted as a DRW (discussed in Section 2.3), then the number of Rabin pairs of $P$ is the same as the number of Rabin pairs of $R$. (Specifically, given an NBW with $n$ states then Safra’s procedure generates a DRW with $12^n n^{2n}$ and a Rabin condition with $n$ pairs. While Piterman’s determination procedure generates a DPW with $n^{2n+2}$ and a nongeneralized parity condition with $2n$ colors.)

2. As mentioned earlier, the parity acceptance condition $\pi$ of $P$ can be simplified to a new parity acceptance condition $\pi'$ with at most $2 \times R_I(\mathcal{L})$ colors through the procedure of Carton and Maceiras [CM99]. There may exist a smaller DPW which accepts $\mathcal{L}$ but it cannot have fewer colors than $\pi'$. On the other hand, given $R$ the first problem of computing $R_I(\mathcal{L})$ through it is an NP-hard problem [KPBV95]. Furthermore, the procedure proposed by the authors to obtain another Rabin automaton $R'$ with $R_I(\mathcal{R})$ pairs from $R$ incurs a state explosion. In this regard, $P$ is superior to $R$.

3. A direct consequence of the previous point is that LTL formulae which describe safety properties but fail the syntactic check (discussed in Section 2.6.1) can be identified. The property is converted to a DPW which is then optimized using the algorithm of Carton and Maceiras. If the resulting DPW can be expressed as a DBW and there are no transitions from the non-accepting states to accepting states then the DPW/DBW is a safety automaton.

Further improvements in Piterman’s determination procedure were independently proposed. In [Sch09], Schewe used history trees to improve the complexity of Piterman’s procedure while in [LW09], Liu and Wang used compact Safra trees to improve the complexity of Piterman’s procedure. Given an NBW with $n$ states, Piterman’s determination procedure produces a DPW with $O(n^n n!)$ states while the new procedures produce a DPW with $O(n!^2)$ states.

The conversion of NBW to a DPW through Piterman’s construction does not guarantee minimality in terms of number of states or the Rabin index of the language accepted by the NBW. But the complexity
of the acceptance condition can be rectified by using the algorithm of Carton and Maceiras. As mentioned in Section 2.6, a DBW can be obtained for a recurrence property, but identifying an LTL formula as a recurrence property is EXPTIME and PSPACE-hard [KV98].

In practice, Piterman’s determinization procedure is computationally expensive and not suited for efficient symbolic implementation (briefly discussed in Section 3.11). Therefore, determinization of moderately large NBW should be avoided whenever possible. For instance, when checking realizability of a set of \( \omega \)-regular properties, then each property should be converted to a deterministic automaton and then these automata should be composed with each other. Finally, the two-player game can be derived from the resulting deterministic automaton.

It can now be concluded that determinization is a computationally expensive step and contributes to the exponential blow up in the number of states of the NBW. Therefore, one should adopt approaches that will minimize the negative impact of the blow up resulting from the determinization procedure. In the next section, it is discussed that NBW obtained from LTL formulae can be determinized through a simpler determinization procedure.

### 3.8 Specialized Determinization Procedure

Piterman’s determinization procedure can generate a DPW for any NBW. It has been briefly discussed in Section 2.5 through the Example 2.5.1 that there exist \( \omega \)-regular languages that cannot be expressed by any LTL formula. This leads one to wonder if there exists a simpler determinization procedure for an NBW that has been obtained from an LTL formula \( \varphi \). This question was answered in [Sch04, MSL08], where it was observed that one can exploit the temporal hierarchy (discussed in Section 2.6.1) of LTL formulae. The NBWs of LTL formulae that satisfy the syntactic characterization of either the persistence or the recurrence class can be determinized through the breakpoint construction (an extension of the subset construction [Sco59]). Since every LTL formula \( \varphi \) has an equivalent formula \( \psi \) that is a Boolean combination of formulae from the persistence and recurrence classes, a deterministic automaton for \( \varphi \) can be obtained by composing the automata of individual formulae of \( \psi \) and then generating a Rabin or Streett acceptance condition (counters may be required to obtain a Rabin or Streett acceptance condition).
There exist cases when it is prohibitively expensive to convert an arbitrary LTL formula $\varphi$ to an equivalent formula which is a Boolean combination of persistence and recurrence formulae. In such cases the breakpoint construction cannot be used. However a procedure was proposed in [Sch04, MSL08] to convert any LTL formula to an unambiguous NBW.

**Definition 3.8.1.** An $\omega$-automaton is **unambiguous** when a word accepted by the automaton may have an arbitrary number of non-accepting runs but only one accepting run.

When an NBW obtained from an LTL formula is unambiguous then a more efficient determinization procedure exists [MS10]. This determinization procedure unlike Piterman’s and Safra’s procedures does not rely on Safra trees. This specialized procedure also converts NBW to a nongeneralized DPW.

In [KC11], the authors present a unified determinization approach through which one can obtain a deterministic automaton from either a deterministic or a nondeterministic $\omega$-automaton with any type of acceptance condition. The complexity bounds of the deterministic automata are comparable to the determinization procedures discussed in Section 3.7. Even though this unified approach is not practically feasible, it is important from a theoretical standpoint.

The determinization procedures briefly discussed in this section do not offer better bounds in terms of the size of the deterministic parity automaton, and there is little evidence that these approaches will perform better than Piterman’s determinization procedure in practice. (If the property is syntactically safe then the determinization procedure based on subset construction [Sco59] is the preferred choice.) Therefore, in the tool based on the ideas proposed in this thesis Piterman’s determinization procedure is still preferred over the other determinization approaches.

The above is only true when the property is not syntactically safe. When the property passes the syntactic check of a GR(1) property then only the individual components are determinized through the determinization procedure. For instance the following property is a syntactic GR(1) property

$$G(L \rightarrow XF \neg b) \land GF h \rightarrow GF(\neg b_0 \lor \neg m) \land GF(\neg b_1 \lor m).$$

In this property only the recurrence property $GF(L \rightarrow XF \neg b)$ is determinized through the Piterman’s construction. The remaining properties are of the form $GF p$. Whenever the tool detects this property then it
generates the corresponding DBW (this property is the most common type of recurrence property). Finally, the automata from each property are composed with each other. It has already been discussed in Section 2.4 how to generate a nongeneralized parity condition for such a property. Alternatively, the specialized technique to solve games obtained from such properties has already been discussed in Section 3.3. Another efficient approach to solve the game obtained from this property has been discussed in Chapter 4.

3.9 Solving Parity Games

It was briefly mentioned in Section 2 that during synthesis of a reactive system from its specification, an \( \omega \)-automaton is obtained from the specification. Then, a two-player game is derived from this automaton. This section briefly discusses algorithms that can be used to solve nongeneralized, conjunctive, or disjunctive parity games.

3.9.1 Nongeneralized Parity Games

Nongeneralized parity games can be solved by Zielonka’s algorithm described in [Zie98, JPZ06] shown in Figure 3.7. This section provides an overview of this algorithm along with the computation of winning strategies for each player. A more detailed, highly readable exposition can be found in [JPZ06]. Given a game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) with \( n = |Q| \) and \( k \) as the highest priority of any state \( q \in Q \), then this algorithm runs in time \( O(n^k) \). The more recent algorithms for solving parity games [Jur00, JPZ06, Sch07] have better bounds, but are not suitable for efficient symbolic implementation, which is discussed in more detail in Section 3.11. All of these algorithms generate a subset of positional strategies for either player, as nongeneralized parity games admit positional strategies [Tho95].

A key step in the algorithm of Fig. 3.7 is the computation of the \( i \)-attraction in \( G \) of a set of vertices \( A \), that is, the set of vertices \( \text{attr}_i(G, A) \) of \( G \) from which Player \( i \) can force a visit of some vertex in \( A \).

**Definition 3.9.1.** Given a turn-based game \( G = (Q, Q_0, Q_1, \delta, \alpha) \) and the set of states \( A \subseteq Q \), the set \( A_i \subseteq Q \) is the \( i \)-attraction of \( A \) in \( G \) if it is the least set such that

\[
(1) \quad A \subseteq A_i,
\]
algorithm Zielonka($G$)
if $G = (\emptyset, \emptyset)$ then return $(\emptyset, \emptyset, \emptyset, \emptyset)$
d $\leftarrow d(G)$; $A \leftarrow A_d(G)$
i $\leftarrow d \mod 2$; $j \leftarrow \neg i$
($A_i, \Delta_i$) $\leftarrow \text{attr}_i(G, A)$
($U_0, U_1, T_0, T_1$) $\leftarrow \text{Zielonka}(G \setminus A_i)$
if ($U_j = \emptyset$) then
  $U_i \leftarrow V(G)$
  $T_i \leftarrow \Delta_i \cup T_i \cup \{(u, v) \in E(G) \mid u \in A \cap Q_i\}$
else
  ($A_j, \Delta_j$) $\leftarrow \text{attr}_j(G, U_j)$
  ($U_0, U_1, W_0, W_1$) $\leftarrow \text{Zielonka}(G \setminus A_j)$
  $U_j \leftarrow V(G) \setminus U_i$
  $T_i = W_i$
  $T_j = T_j \cup \Delta_j \cup W_j$
fi
return ($U_0, U_1, T_0, T_1$)

attr$_i(G, A) = A \cup \{q \in Q_j \mid \forall \sigma \in \Sigma . \delta(q, \sigma) \in A\} \cup \{q \in Q_i \mid \exists \sigma \in \Sigma . \delta(q, \sigma) \in A\}$

Figure 3.7: Zielonka’s algorithm for nongeneralized parity games. $d(G)$ is the largest priority of $G$. $A_d(G)$ is the set of states of $G$ assigned the priority $d$.

(2) $\forall q \in (Q_i \setminus A_i). \delta(q) \subseteq (Q \setminus A_i)$ and

(3) $\forall q \in (Q_{\neg i} \setminus A_i). \delta(q) \not\subseteq A_i$.

The computation of attr$_i(G, A)$ is a least fixpoint computation, in which $A_i$ is initialized to $A$ and the set of attraction transitions $\Delta_i$ is initially empty. Vertices of $G$ in $Q_i$ (controlled by Player $i$) are added if they have at least one transition into a vertex already acquired to attr$_i(G, A)$, while vertices of $G$ in $Q_j$ are added if all their transitions lead to vertices already acquired to attr$_i(G, A)$. When a vertex in $Q_i$ is added to $A_i$, the transitions from it to vertices already in attr$_i(G, A)$ are added to $\Delta_i$.

At Line 6, the algorithm recurs on the subgame induced by the removal of states in $A_i$ from the game $G$.

Definition 3.9.2. A turn-based game $G^S = (Q^S, Q_0^S, Q_1^S, \delta^S, \Pi^S)$ is the subgame of turn-based game $G = (Q, Q_0, Q_1, \delta, \Pi)$ induced by $Q^S \subseteq Q$ if

(1) $Q_i^S = Q_i \cap Q^S$, 

(2) $\forall q \in (Q_i \setminus A_i). \delta(q) \subseteq (Q \setminus A_i)$ and

(3) $\forall q \in (Q_{\neg i} \setminus A_i). \delta(q) \not\subseteq A_i$. 

The computation of attr$_i(G, A)$ is a least fixpoint computation, in which $A_i$ is initialized to $A$ and the set of attraction transitions $\Delta_i$ is initially empty. Vertices of $G$ in $Q_i$ (controlled by Player $i$) are added if they have at least one transition into a vertex already acquired to attr$_i(G, A)$, while vertices of $G$ in $Q_j$ are added if all their transitions lead to vertices already acquired to attr$_i(G, A)$. When a vertex in $Q_i$ is added to $A_i$, the transitions from it to vertices already in attr$_i(G, A)$ are added to $\Delta_i$.

At Line 6, the algorithm recurs on the subgame induced by the removal of states in $A_i$ from the game $G$.

Definition 3.9.2. A turn-based game $G^S = (Q^S, Q_0^S, Q_1^S, \delta^S, \Pi^S)$ is the subgame of turn-based game $G = (Q, Q_0, Q_1, \delta, \Pi)$ induced by $Q^S \subseteq Q$ if

(1) $Q_i^S = Q_i \cap Q^S$, 

(2) $\forall q \in (Q_i \setminus A_i). \delta(q) \subseteq (Q \setminus A_i)$ and

(3) $\forall q \in (Q_{\neg i} \setminus A_i). \delta(q) \not\subseteq A_i$. 

The computation of attr$_i(G, A)$ is a least fixpoint computation, in which $A_i$ is initialized to $A$ and the set of attraction transitions $\Delta_i$ is initially empty. Vertices of $G$ in $Q_i$ (controlled by Player $i$) are added if they have at least one transition into a vertex already acquired to attr$_i(G, A)$, while vertices of $G$ in $Q_j$ are added if all their transitions lead to vertices already acquired to attr$_i(G, A)$. When a vertex in $Q_i$ is added to $A_i$, the transitions from it to vertices already in attr$_i(G, A)$ are added to $\Delta_i$.

At Line 6, the algorithm recurs on the subgame induced by the removal of states in $A_i$ from the game $G$.
\((2) \ \forall u,v \in Q^S, v \in \delta^S(u) \leftrightarrow v \in \delta(u) \text{ and}\)

\((3) \ \Pi^S \text{ is the restriction of } \Pi \text{ to } Q^S.\)

The notion of a subgame of a larger game described here is central to understanding Zielonka’s algorithm. Let \(G = <\Sigma, Q, Q_0, Q_1, \delta, \Pi>\) be a turn-based parity game, where \(\Pi\) is a singleton (\(\Pi = \{\pi}\) and \(\pi\) is a simple parity condition). The subgame induced by the set of states \(Q_s \subseteq Q\) is given by \(G_s = <\Sigma, Q_s, Q_s \cap Q_0, Q_s \cap Q_1, \delta, \Pi>\). The notation \(G \setminus A\) refers to the subgame induced by the set of states \(Q \setminus A\).

For the parity game \(G\), Zielonka’s algorithm can be summarized by the following observations:

- The algorithm computes the states \(A_i\) that can be attracted by Player \(i\) to states \(A_d(G)\) with the highest priority \(d\) in \(G\), where \(i = d \mod 2\).
- The algorithm then finds the winning states \(W = U_i\) and losing states \(L = U_j\) for Player \(i\) in the game \(G \setminus A_i\). Here, \(j = 1 - i\).
- The states in \(L\) are losing for Player \(i\) in \(G\) because it cannot attract them to either \(W\) or \(A_i\).
- The algorithm computes the states \(A_j\) that can be attracted to \(L\) in \(G\) by Player \(j\). The states \(A_j \cup L\) are losing for Player \(i\) in \(G\).
- The algorithm recurs on \(G_r = G \setminus A_j\). Let \(W_r\) and \(L_r\) be the winning and losing states in \(G_r\) for Player \(i\). Then \(W_r\) and \(L_r \cup A_j\) are the winning and losing states in \(G\) for Player \(i\).
- The winning strategy of Player \(i\) in \(W_r\) is computed in the second recursive call. The winning strategy of Player \(j\) in \(L_r\) is computed in the second recursive call. Player \(j\)’s strategy for states in \(L\) is computed in the first recursive call, while its strategy for states in \(A_j \setminus L\) is computed through the second attraction computation.
- In the case when \(L = \emptyset\) then the algorithm declares \(W \cup A_i\) as the winning states of \(G\). The winning strategy of Player \(i\) for states in \(W\) is computed in the first recursive call, while the strategy...
Figure 3.8: A parity game. Priority of states are shown in parentheses

for states in $A_i \setminus A_d(G)$ is computed in the first attraction computation; the states in $A_d(G)$ impose no obligation and can be followed any state in $G$.

The following example describes how Zielonka’s algorithm computes the winning strategies during an attraction computation.

**Example 3.9.3.** Consider the game shown in Figure 3.8. It is a turn-based parity game with two priorities. Zielonka’s algorithm returns all vertices of the graph in $U_1$. The bold arrows denote transitions in $T_1$. Since both transitions out of vertices $d$ and $g$ are in $T_1$, there are four distinct memoryless winning strategies for Player 1. Note that transitions $(e, b)$ and $(f, c)$ are not in $T_1$. A strategy that used both of them would have to use memory to prevent the play from cycling through $b$, $f$, $c$, and $e$ without visiting $a$. Exclusion of $(e, b)$ and $(f, c)$ from $T_1$ prevents such cycles, but also rules out eight memoryless strategies that include one but not the other. The computation of attractions imposes a pre-order on the vertices in the fixpoint: $v \leq u$ iff $v$ is added to $\text{attr}_i(G, A)$ no later than $u$. (It is a pre-order, because breadth-first computation of the attraction may add multiple vertices simultaneously.) A transition $(u, v)$ between two vertices of $\text{attr}_i(G, A)$ is in $T_1$ iff $v < u$.

The following example describes how Zielonka’s algorithm computes the winning states for each player on a small example.

**Example 3.9.4.** Consider the game shown in Figure 3.9. It is a turn-based parity game with four priorities. Zielonka’s algorithm returns all vertices as winning for Player 1. There are four invocation of
Zielonka’s algorithm on various subgames (excluding invocation on empty games). In the following description \([q_1, q_3, q_4]\) denotes a game graph induced by the states \([q_1, q_3, q_4]\) (this game has only four edges \(q_4 \to q_3, q_3 \to q_1, q_1 \to q_4\) and \(q_1 \to q_4\)).

(1) \(G_1 = [q_0, q_1, q_2, q_3, q_4]\): In this game the highest color is \(3\). Therefore, \(A_3 = \{q_4\}\) and \(\text{attr}_1(G_1, A_3) = A_3\). Then the algorithm recurs on \(G_2 = [q_0, q_1, q_2, q_3]\).

(2) \(G_2 = [q_0, q_1, q_2, q_3]\): In this game the highest color is \(2\). Therefore, \(A_2 = \{q_3\}\) and \(\text{attr}_0(G_2, A_2) = \{q_2, q_3\}\). Then the algorithm recurs on \(G_3 = \{q_0, q_1\}\).

(3) \(G_3 = [q_0, q_1]\): In this game the highest color is \(1\). Therefore, \(A_1 = \{q_1\}\) and \(\text{attr}_0(G_3, A_1) = \{q_0, q_1\}\). Then the algorithm returns for the first time with \(U_1 = \{q_0, q_1\}\) and \(U_0 = \emptyset\) when the game is \(G_2\). Therefore, the algorithm has to take the else branch because \(U_1 \neq \emptyset\) as in this game \(i = 0\) and \(j = 1\). Finally, \(\text{attr}_1(G_2, U_1) = \{q_0, q_1, q_3\}\) and then the algorithm recurs on \(G_4 = \{q_2\}\).

(4) \(G_4 = \{q_2\}\): In this game the highest color is \(1\). Therefore, \(A_1 = \{q_2\}\) and \(\text{attr}_1(G_4, A_1) = A_1\) Then the algorithm returns for the second time with \(U_0 = \emptyset\) and \(U_1 = \{q_2\}\) when the game is \(G_2\).

The algorithm returns the third time with \(U_0 = \emptyset\) and \(U_1 = \{q_0, q_1, q_2, q_3\}\) with the game being \(G_1\). The algorithm returns one more time with \(U_0 = \emptyset\) and \(U_1 = \{q_0, q_1, q_2, q_3, q_4\}\) to the original invocation to Zielonka’s algorithm.

In this example, the else branch was only taken when the algorithm is invoked on the game \(G_2\).

The memoryless strategy computed by Zielonka’s algorithm is \(q_3 \to q_1, q_0 \to q_1\). There also exist memoried strategies which the algorithm does not compute. For instance, in the above example, a memoried strategy is that if the \(q_3\) was visited from \(q_4\) then \(q_3\) can also visit \(q_2\). Alternatively, \(q_3\) can visit \(n\) times before it visits \(q_1\).

Game solving algorithms based on attraction computations—not just Zielonka’s—therefore only compute a subset of all memoryless strategies. Even in algorithms not explicitly based on attraction computations like the one of [Jur00, JPZ06, Sch07], strategy computation relies on the order in which vertices
are added to the set of winning positions. Therefore, these algorithms compute a subset of memoryless strategies. Even though Zielonka’s algorithm has been used to illustrate that only a subset of memoryless strategies are computed, this observation applies more generally to Chatterjee’s algorithm [CHP07] for disjunctive parity games as well (disjunctive parity games admit memoryless strategies however the algorithm only finds a subset).

The authors of [BJW02] study a partial permissivity ordering on nondeterministic strategies. The authors declare a strategy as permissive if it allows the player the most freedom while guaranteeing that it is still a winning strategy. They show that safety games have maximal strategies that are memoryless, and that every game that has a maximal memoryless strategy is equivalent to a safety game. For other types of games, they show that a permissive strategy cannot be computed from the game directly. However, they show how to convert an arbitrary parity game to a safety game (which is significantly larger) such that a permissive strategy for the original game can be obtained by playing this safety game. The permissive strategy is not memoryless with respect to the original game and the memory present in the permissive strategy is bounded by the size of the original game.

Finally, the performance of various parity game solving algorithms was done in [FL09]. It was reported that Zielonka’s algorithm outperforms other algorithms. The authors made this observation after measuring the empirical performance of various parity game solving algorithms on both synthetic and do-

Figure 3.9: A parity game with four colors: Priority of states are shown in parentheses. The circle player wins.
main specific games.

3.9.2 Conjunctive and Disjunctive Parity Games

This section focuses on solving conjunctive parity games. These are important to discuss because specification is often a conjunction of \( \omega \)-regular properties and such a specification is easily converted to a parity game with a conjunctive winning condition. The system needs to guarantee all the winning conditions while the adversary only needs to win one. Since the goal is to synthesize the system and not the environment, the focus is on solving conjunctive parity games.

Conjunctive parity games and disjunctive parity games are duals of each other as discussed in Section 2.3. In a conjunctive (disjunctive) parity game, the protagonist is the conjunctive (disjunctive) player as it has to satisfy the conjunctive (disjunctive) parity condition, while the antagonist is the disjunctive (conjunctive) player. These games admit positional strategies for the disjunctive player while the conjunctive player requires finite memory for its strategies [Tho95].

Conjunctive (disjunctive) parity games can be solved by Chatterjee’s algorithm described in [CHP07]. This algorithm, shown in Figure 3.10, is an extension of Zielonka’s algorithm. The correctness of the algorithm in Figure 3.10 is proved in [CHP07]. The following discussion is a summary of how the winning strategies in a conjunctive parity game for the protagonist and antagonist are computed for their respective winning regions. The reader is encouraged to consult [CHP07] for a detailed discussion on finding the winning regions of each player in a conjunctive parity game.

The protagonist requires finite memory for its winning strategy in a conjunctive parity game. The memory is required so that once the objectives a particular parity condition are fulfilled then one can rotate to another objective defined by the remaining parity conditions. Once all the objectives defined by the different parity conditions have been fulfilled, the whole process starts over again. The following examples informally describe how the algorithm computes the winning strategies for conjunctive parity game.

Example 3.9.5. Consider the parity game shown in Figure 3.11 with the conjunctive parity condition \( \{\pi_1, \pi_2\} \), where \( \pi_1 \) assigns priority 1 to b and priority 0 to the other states, while \( \pi_2 \) assigns priority 1 to c
algorithm `CONJParityWin(G)`
  1. if $G = (\emptyset, \emptyset)$ then return $(\emptyset, \emptyset, \emptyset)$
  2. for each $\pi_i \in \{\pi_1, \pi_2, \ldots, \pi_k\}$
     1. $j \leftarrow 0$
     2. $G_1 \leftarrow G \setminus attr_1(G, MAX(G, \pi_i, 1))$
     3. $H_1 \leftarrow G_1 \setminus attr_0(G_1, MAX(G_1, \pi_i, 0))$
     4. do
        1. $j \leftarrow j + 1$
        2. $C_j \leftarrow \text{CONJParityWin}(H_j)$
        3. $C_j^\text{attr} \leftarrow attr_1(G_j, C_j)$
        4. $G_{j+1} \leftarrow G_j \setminus C_j^\text{attr}$
        5. $H_{j+1} \leftarrow G_{j+1} \setminus attr_0(G_{j+1}, MAX(G_{j+1}, \pi_i, 0))$
     4. while $G_{j+1} \neq G_j$
     5. if $G_{j+1} \neq (\emptyset, \emptyset)$
        1. $D \leftarrow attr_0(G, G_{j+1})$
        2. $C \leftarrow \text{CONJParityWin}(G \setminus D)$
        3. return $C$
     6. end
  3. end for each
  4. return $A(G)$
  5. end

$\text{MAX} (G, \pi, P)$
  1. $d \leftarrow d(G, \pi)$
  2. do
     1. $A \leftarrow A_d(G, \pi)$
     2. if $A = \emptyset$ then $d \leftarrow d - 1$
     3. else break
     4. while $d \geq 0$
     5. if $d \mod 2 \neq P$ then return $\emptyset$
     6. else return $A$
  3. end

Figure 3.10: Chatterjee’s algorithm for conjunctive and disjunctive parity games. $A(G)$ is the set of states of game $G$. $d(G, \pi)$ is the highest priority of $G$ with respect to $\pi$. $A_d(G, \pi)$ is the set of states assigned the priority $d$ by $\pi$ in the game $G$.

and priority 0 to the other states. Player 1 moves from state $a$. (That is, $Q_0 = \{b, c\}$ and $Q_1 = \{a\}$.) Note that the parity conditions are effectively Büchi conditions corresponding to the LTL formula $G F b \land G F c$.

Suppose the algorithm of Figure 3.7 is used to compute the winning positions according to $\pi_1$. For the graph $G$ in Figure 3.11, the attractor of $\{b\}$ for Player 1 is computed as follows at Line 5. Initially $b$
is in the attractor and the set of transitions is empty; $a$ is then added to \( \text{attr}_1(G, \{ b \}) \) and the transition from $a$ to $b$ is added to $\Delta_1$. Finally $c$ is added to the attractor and the transition from $c$ to $a$ is added to the set of transitions. Since $G \setminus A_1$ is empty, the recursive call returns immediately with $U_0 = \emptyset$. Therefore all positions are winning for Player 1. The transition from $b$ to $a$ is added to $\Delta_1$ to produce $T_1$. It is clear that there is no winning strategy for $\pi_2$ when only transitions from $T_1$ are allowed from states in $Q_1$, though there exist strategies to win both $\pi_1$ and $\pi_2$ on $G$. One such strategy uses one bit of memory to alternate between the two transitions out of $a$. In this particular case, Chatterjee’s algorithm is also able to synthesize a winning strategy with a single memory bit.

**Example 3.9.6.** In the conjunctive parity game with two parity winning conditions shown in Figure 3.12, the disjunctive player (antagonist) wins. This player controls the states represented by squares. The conjunctive player (protagonist) controls the circle states.

Notice that the protagonist wins all the nongeneralized parity games when each parity condition is considered individually. For instance, if the first condition is considered then the protagonist has the strategy $q_3 \rightarrow q_1$ and $q_0 \rightarrow q_1$. If the antagonist choses to visit $q_4$ from any of its states then it loses because the highest recurring priority is odd, on the other had if it does not chose to visit $q_4$ then it is trapped in the loop between $q_1$ and $q_0$.

Similarly, when the second condition is considered then the protagonist has the strategy $q_3 \rightarrow q_2$ and $q_0 \rightarrow q_3$. In this case as well, if the antagonist chooses to visit $q_4$ then the highest recurring priority is odd. If it does not choose this strategy then it is trapped in the loop between $q_3$ and $q_2$.

However, if the conjunctive parity condition is considered then the protagonist cannot trap the antagonist in either $\{q_3, q_2\}$ or $\{q_1, q_0\}$, because neither of these strategies are winning for the other parity condition.

After the protagonist finds the states that it can control to $\{q_4\}$ (because it is the highest priority which
is odd with respect to both conditions), the protagonist searches for a subset of remaining states where it can trap the antagonist such that all the parity conditions are satisfied. If the protagonist focused on the first parity condition, the only trap that existed with respect to the first condition is \{q_1, q_2\} but this is losing and therefore the protagonist loses the game because the antagonist can willingly get trapped here and win the game.

If the protagonist focused on the second condition, the only set it could trap the antagonist in is \{q_3, q_2\}. However, this is losing with respect to the first condition. Therefore, the protagonist loses the game.

The antagonist can let the protagonist try to compute the winning strategies and if it fails it can use that information to compute its own. In this game the antagonist’s strategy is q_4 \rightarrow q_3, q_2 \rightarrow q_4, q_1 \rightarrow q_0.

**Example 3.9.7.** In the conjunctive parity game with two parity winning conditions shown in Figure 3.12, the disjunctive player (antagonist) loses. This player controls the states represented by squares. The conjunctive player (protagonist) controls the circle states.

The protagonist computes the set of states that can be controlled to highest priority which is odd. When its focus is the first parity condition then this set is \{q_4\}. In the remaining states, the protagonist finds the trap \{q_1, q_0\}. In this trap, the protagonist is still able to win the second condition. The antagonist an try
Figure 3.13: A conjunctive parity game with both conditions have four colors: Priority of states are shown in parentheses. The conjunctive player which controls the circle states wins.

to leave the trap but it will have to visit $q_4$ and when it comes back to $q_2$ the protagonist again tries to trap it in $\{q_1, q_0\}$. Whether the antagonist choses to leave the trap or stay stuck in it, the highest recurring priority is odd.

Now the protagonist has to focus on the second parity condition. It computes the set of states that can be controlled to the highest priority which is odd, in this case it is $\{q_0\}$. The protagonist now finds the trap $\{q_4, q_3, 2_1\}$. If the antagonist tries to leave this trap then it ends up visiting $q_0$. Whether the antagonist choses to leave the trap or stay stuck in it, the highest recurring priority is odd.

The winning strategy computed by the Chatterjee’s algorithm is as follows. There are three games $G = [q_4, q_3, q_2, q_1, q_0], G_1 = [q_3, q_2, q_1, q_0], G_2 = [q_4, q_3, q_2, q_1]$, where the second and third games are subgames of the first game. The memory needed by the strategy is represented as a pair. The first element represents if the objective of the first game has been satisfied. In other words, if the current state of the memory is $S_1$ then the sate $q_4$ has not been visited and the antagonist is trapped in $\{q_1, q_0\}$. If the memory is $S_2$ then the state $q_0$ has not been visited and the antagonist is trapped in $\{q, 4, q_3, q_1\}$. The second element of the pair is to remember the objectives of the subgame.

It is important to remember that $\{q_4\}$ is the objective in $G$ for the first parity condition. It is also the objective in $G_2$ for the first parity condition. Similarly, $\{q_0\}$ is the objective in $G$ for the second parity condition.
condition. It is also the objective in $G_1$ for the second parity condition. In $G_1$ the objective of the first parity condition is $\{q_1\}$, while in $G_2$ the objective of the second parity condition is the set $\{q_3, q_4\}$.

- $((S_1, X), q_4) \rightarrow ((S_2, S_1), X)$: The objective of the game $G$ w.r.t. first parity condition was fulfilled and the memory is updated to seek the objective w.r.t. second parity condition in $G$. (The state is not controlled by the protagonist so only the memory is updated.)

- $((S_2, S_1), q_4) \rightarrow ((S_2, S_2), X)$: The objective of the game $G_2$ was fulfilled w.r.t. first parity condition and the memory is updated to seek the objective w.r.t. second parity condition in $G_2$.

- $((S_2, X), q_0) \rightarrow ((S_1, S_1), q_1 \text{ or } q_3)$: The objective of the game $G$ w.r.t. second parity condition was fulfilled and the memory is updated to seek the objective w.r.t. first parity condition in $G$.

- $((S_1, S_1), q_1) \rightarrow ((S_1, S_2), X)$: The objective of the game in $G_1$ w.r.t. first parity condition was fulfilled and the memory was updated to seek the objective w.r.t. second parity condition in $G_1$.

- $((S_1, S_2), q_0) \rightarrow ((S_1, S_1), q_1)$: The objective of the game in $G_1$ w.r.t. second parity condition was fulfilled and the memory was updated to seek the objective w.r.t. first parity condition in $G_1$.

- $((S_2, S_2), q_3) \rightarrow ((S_2, S_1), q_1)$: The objective of the game in $G_2$ w.r.t. second parity condition was fulfilled and the memory was updated to seek the objective w.r.t. first parity condition in $G_2$.

The above list only shows the transitions in which there is a change in the memory state. All the other transitions do not change the state of the memory.

If the right objective of $G$ is fulfilled then the memory of the subgames is reset as the antagonist escaped the trap and keeping track of the obligations of the subgame is not required anymore.

It is important to note that in the Example 3.9.7, the conjunctive player had a memoryless strategy. This strategy is $q_3 \rightarrow q_1$ and $q_0 \rightarrow q_1$. The extraction of a simple strategy from the one computed by Chatterjee’s algorithm is not a trivial task. The complexity of the strategy relation generated by this algorithm increases dramatically as the number of parity conditions grows. Therefore, the use of this algorithm is avoided whenever possible.
In the synthesis framework described in this thesis, Chatterjee’s algorithm is only used to solve games derived from General Reactive \( (m) \) properties (defined in Section 2.6.1). It is discussed in Chapter 4 that games derived from General Reactive \( (1) \) properties or games with general Streett\((1)\) conditions can be solved by iterating Zielonka’s algorithm without converting the game to a more complex game with a non-generalized parity winning condition.

The performance of Chatterjee’s algorithm for conjunctive (disjunctive) parity games can be improved by invoking procedures that simplify the conjunctive (disjunctive) parity condition. Particularly, when the algorithm recurs on a subgame of \( G \), then each parity acceptance condition can be simplified by invoking the procedure of Carton and Maceiras and the new conjunctive (disjunctive) parity condition can then be simplified by invoking Theorem 4.2.1. Although the suggested improvements do not affect the worst case behavior of Chatterjee’s algorithm, they can yield significant performance improvements in practice.

### 3.10 Safraless Approaches

An alternative to the game-based approach to synthesis discussed in the previous sections of this chapter was developed in [KV05, KPV06]. The motivation behind this approach was to avoid the expensive determinization procedure of Safra and its variants; that is why this approach is also known as **Safraless decision procedure**. An LTL formula \( \varphi \) is negated and then converted to an NBW. The NBW is negated and thus a UCW is obtained. The language accepted by the automaton UCW is \( L(\varphi) \). Instead of deriving a two-player game from UCW, a UCT is derived from the UCW. A tree automaton accepts a language on infinite trees. The realizability of \( \varphi \) is established if the language accepted by UCT is non-empty, otherwise \( \varphi \) is unrealizable. The complexity of playing a game derived from a UCW is identical to checking language emptiness of a UCT derived from the same UCW. In practice, the language emptiness of the UCT \( A \) is checked by converting it to an NBT \( A' \) such that the reduction guarantees the following:

\[
L(A') \subseteq L(A) \quad \text{and} \quad L(A) \neq \emptyset \rightarrow L(A') \neq \emptyset .
\]

Finally, the non-emptiness of \( A' \) is checked to prove the realizability of \( \varphi \). An infinite tree in the language of \( A' \) is then used to generate an implementation of \( \varphi \). Intuitively this infinite tree is related to the winning
strategy of the protagonist in the two-player game derived from the UCW. The following techniques, which are variants of this approach, attempt to generate an efficient implementation for $\varphi$. (The efficiency of the implementation is related to the combinational and sequential logic that is required by the hardware implementation.)

The bounded synthesis approach inspired by bounded model checking (BMC) [BCC+99] was proposed in [SF07]. This bounded synthesis approach is based on the Safraless decision procedure discussed earlier in this section. Several variants of the bounded synthesis approach have been developed. The basic idea of these approaches is to produce a UKCW from the UCW by adding a non-decreasing counter of bound $K$ to each state. The number of visits of the set of rejecting states of the UCW are recorded by this counter. Notice that a UKCW accepts a safety language. A run of the UCW that visits the set of rejecting states more than $K$ times is rejected by the safety automaton UKCW. The authors prove that the language accepted by the UKCW is a subset of the language accepted by the UCW and, for some $K > 0$, the UKCW and the UCW accept the same language.

Originally the objective of the approach of [SF07] was to synthesize distributed controllers. As mentioned earlier, this problem in its full generality is undecidable. In the case when the distributed controller is based on a fully informed architecture the synthesis problem can be solved. A special input-preserving deterministic automaton $D_I$ is created that reduces the problem of realizability to the one of satisfiability. The specification of each individual controller is converted to a UCW and for some bound $K$, safety automaton UKCW are constructed. The value of $K$ is bounded by the transition structure of the UCW and the number of its acceptance conditions. The UKCWs of all controllers are composed and the resulting safety automaton is composed with $D_I$. If the language of the resulting automaton is not empty then the specification is realizable and an implementation can be obtained. In practice, a constraint system based on uninterpreted functions is used to represent the composition of UKCWs and $D_I$ efficiently. The language emptiness of the composition can be checked by unraveling the transition relation $K$ times and checking for a satisfying assignment using a Satisfiability solver. If the satisfying assignment for some $K > 0$ exists, then it is used to generate an implementation of the distributed controller.

The bounded synthesis approach was adopted by [FJR09, FJR10, Ehl10], where it was applied to the
synthesis of centralized controllers in the game-theoretic setting. In [FJR09, FJR10], the UKCW is interpreted as a game between the system and the environment and solved using an antichain-based algorithm. An implicit representation is used for the safety game UKCW, and that is why the antichain-based algorithm is necessary for the efficiency of this approach. On the other hand, the approach of [Ehl10] simplifies the transition relation of the counters by allowing the system to increment the counter by \( n > 0 \) even when a rejecting state is not visited. The system’s objective is to satisfy the safety property; therefore, this relaxation only simplifies the transition structure of the UKCW. Another simplification is to compute the strongly connected components (SCCs) of the UKCW, and the transient states are identified; the system is given the freedom to reset the counters associated with these transient states. Finally, the symbolic encoding of UKCW is carefully selected and the safety game UKCW is solved.

**Example 3.10.1.** Let \( \varphi = G F p \) be an LTL property. To obtain a UCW for this property, \( \varphi \) is negated, where \( \neg \varphi = F G \neg p \). The NBW for \( \neg \varphi \) is obtained using the standard LTL translation. The NBW is interpreted as a UCW and both automata are shown in Figure 3.14. One can observe that \( L(UCW) = L(\varphi) \).

The UKCW for \( K = 2 \) is shown in Figure 3.15. This automaton accepts any run that is accepted by the UCW as long as the number of times a non-accepting state of the UCW appears in the run is less than 3. The UKCW is a safety automaton and if the protagonist has a winning strategy in the safety game derived from the UKCW, then the protagonist also has a winning strategy in the non-safety game derived from the UCW.

If the specification is unrealizable then all the bounded synthesis approaches continue to increase the bound and are unable to prove unrealizability. This problem is resolved by having a parallel process where the specification is negated. A UCW is obtained for the negated specification. From the UCW a sequence of UKCW are obtained as detailed above; the environment tries to win the safety game UKCW. If the environment wins one of the safety games then the specification is proven to be unrealizable. The two parallel processes stop when either of the players is able to win one of their respective safety games, as the

---

3 Since these games contain a lot of counters, the antichains are used to represent large set of states based on the value of the counters. For instance, a state \((q, k)\) where \( k \) is the value of the counter is used to represent the set of states \( \{(q, k), (q, k+1), (q, k+2), \ldots, (q, K)\} \). This is possible because in a UKCW if a state \((q, k)\) is winning then the states \( \{(q, k+1), (q, k+2), \ldots, (q, K)\} \) are also winning.
Figure 3.14: $\varphi = GFp$. The NBW for $\neg\varphi$ and UCW for $\varphi$. The label of each state reflects the priority of the state. The transitions that are connected (for instance $1 \rightarrow 1 \land 1 \rightarrow 2$) need to be simultaneously taken; this is the reason why the run of a universal automaton is a tree.

other player can never win any of its safety games. This approach is correct because Gale-Stewart games are determinate, which implies that from the initial state of the game only one of the players has a winning strategy.

Figure 3.15: UKCW with $K = 2$ is a safety automaton and $(0, 3)$ is the trap state. The state with the label $\infty$ represent the initial state.
3.11 Efficiency of Symbolic Algorithms

Symbolic algorithms have had significant impact in the field of model checking. Unfortunately, there exist many graph algorithms that are not able to take advantage of symbolic encodings of a graph. This section describes the type of algorithms that are ideally suited for symbolic implementations and the ones that may perform poor on a symbolic representation of a graph.

In Section 2.8, an efficient symbolic encoding of an automaton is discussed. The algorithms that manipulate set of states can be applied on automata with such symbolic encodings. Zielonka’s algorithm for solving nongeneralized parity games and Chatterjee’s algorithm for solving conjunctive parity games are examples of such algorithms. Similarly, model checking algorithms based on $\mu$-calculus in general can be applied to graphs with such encodings.

On the other hand, algorithms that manipulate sets of sets of states cannot use this encoding to represent sets of sets of states symbolically without resorting to some explicit techniques. In such cases, each state of the automaton is assigned a symbolic variable. The set of states $S$ is represented by a minterm, where all the symbolic variables are assigned the value $\top$. Any other minterm represents the subset of $S$ where the state whose symbolic variable is assigned $\top$ is an element of the subset; otherwise it is not an element of the subset. The empty set is represented by the minterm where all the symbolic variables are assigned the value $\bot$.

The subset construction procedure [Sco59], breakpoint procedure for determinization [Sch04, MSL08], and the procedure to generate GFG automata [HP06] are examples of algorithms that manipulate sets of sets of states. It has already been discussed in Section 2.8 that the effectiveness of symbolic algorithms is dependent on finding a good variable order; this in turn is dependent on the number of symbolic variables. Therefore, finding a good variable order for the more elaborate encoding of the automata may often negate the advantages of using symbolic representation of the graph. However, the main problem is that this representation does not scale well.

The parity game solving algorithms of [Jur00, Sch07] are unable to take advantage of the symbolic representation of the graph because these algorithms enumerate the states of the graph. In other words, these
algorithms do not manipulate sets of states; rather, they operate on each individual state of the graph. The parity game solving algorithm of [JPZ06] is also unable to take advantage of the symbolic representation of the graph because this algorithm enumerates a large number of subsets of the states of the parity game. Some of these subsets will not have efficient symbolic representations and will negate the advantages of using symbolic representation of the graph. (When evaluating the performance of parity game solving algorithms in practice, the authors of [FL09] did not include these algorithms in their experimental setup.)

Finally, determinization procedures are difficult to implement symbolically because finding efficient symbolic representation of a tree whose nodes are the set of states of the automaton is not a trivial task. However, in a similar context, practical aspects of using symbolic representations of problems in Weak Second-order Theory of One or Two successors (WS1S/WS2S) has been explored [Kla98].
Chapter 4

The Hierarchical Approach

The first step in a game-based approach to synthesis of an $\omega$-regular specification is to convert the specification into a deterministic automaton that accepts the $\omega$-regular language of the specification. In the next step, a two-player game between the environment (antagonist) and the system (protagonist) is derived from the automaton. Given the set of atomic propositions $X = X_{ed} \cup X_{s} \cup X_{ep}$, the environment assigns values to the variables in the set $X_{ed} \cup X_{ep}$ (the reason for partitioning the variables that are assigned by the environment was discussed in Section 2.7) and the system assigns values to the variables in the set $X_{s}$. If the system has a winning strategy in this game then the specification is realizable, otherwise the specification is unrealizable. An implementation satisfying the specification can be extracted from the winning strategy of the system.

Even though the automaton accepting the language described by the specification can be trivially converted to an input-based game, the approach described in this section can be applied to both turn-based games and input-based games. Input-based games are discussed in detail in Chapter 6, where it will become clear why input-based games may be preferred over turn-based games. Since turn-based games are more commonly studied, the details of this hierarchical approach are discussed in the context of turn-based games. After the discussion of input-based games it will become apparent that the hierarchical approach can also be employed in the context of input-based games after minor adjustments.

Whenever the specification is given by a Boolean combination of $\omega$-regular properties, it is possible to obtain an $\omega$-automaton for each property and then compose the automata. The $\omega$-automata may not be deterministic. Unfortunately, determinization is a very expensive computation as described in Section 3.7.
Therefore, the $\omega$-automaton of each property is determinized and then the deterministic $\omega$-automata (in this thesis DPWs are preferred over other types of deterministic automata) are composed with each other. The eager determinization of $\omega$-automata leads to significant redundancies in the composed automata.

The objective of converting the specification to a deterministic automaton is to obtain a game between the system and the environment. The game is then used find the protagonist’s winning strategies, thus, the existence of protagonist’s winning strategy is a proof that the specification is realizable. The hierarchical approach discussed in this chapter exploits the hierarchy of deterministic $\omega$-automata by adopting an incremental approach to converting the specification to a game; as the game is being constructed by through the composition of deterministic automata it is often possible to simplify the intermediate game representations by removing the states that are easily proven to be losing for the protagonist. (A state that is losing for the conjunctive player in either game is still losing in the product of the two games.) Furthermore, conjunctive parity condition of the composed automaton can often be simplified. In the end, the composed automaton has a simpler conjunctive parity condition and the automaton itself is simpler as it does not contain states that can be easily identified as losing for the protagonist.

4.1 Classification of Deterministic Omega Games

The following definitions describe the games produced by various properties in the Borel hierarchy (discussed in Section 2.6). Since DPWs are preferred in this thesis over other types of deterministic automata, the following concepts are defined in the context of automata with non-generalized parity conditions. These concepts can easily be extended to automata with conjunctive or disjunctive parity conditions. In the remainder of this thesis, the non-generalized parity condition associated with a game is always assumed to be of minimum index unless explicitly stated. (The Rabin index of an $\omega$-regular language has been discussed in Definition 3.7.2. One can obtain a parity acceptance condition with minimal index through the procedure of Carton and Maceiras [CM99].)

Definition 4.1.1. A safety winning condition for a parity game $G = (\Sigma, Q, Q_0, Q_1, \delta, \pi)$ is a function
\[
\pi : Q \rightarrow \{0, 1\}
\]
such that for every state $u \in Q$ such that $\pi(u) = 0$ there is no state $v \in \delta(u)$ such that
\[ \pi(v) = 1. \]

**Definition 4.1.2.** A *guarantee winning condition* for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a function \( \pi : Q \to \{0, 1\} \) such that for every state \( u \in Q \) such that \( \pi(u) = 1 \) there is no state \( v \in \delta(u) \) such that \( \pi(v) = 0 \).

Notice that a safety game can be reduced to an equivalent game with only one state assigned the priority 0. Similarly, the guarantee game can be reduced to an equivalent game with only one state assigned the priority 1 such that neither player has a strategy to escape this state. It may be worth noting that guarantee is co-safety.

**Definition 4.1.3.** An *obligation winning condition* for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a function \( \pi : Q \to \{0, 1\} \) such that every two states \( u \in Q \) and \( v \in Q \) that are reachable from each other satisfy the condition \( \pi(u) = \pi(v) \).

The games with obligation winning condition are obtained from weak automata 2.6.2, while both the safety and guarantee winning conditions are special cases of obligation winning condition as shown in Figure 2.4.

**Definition 4.1.4.** A *persistence* winning condition for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a function \( \pi : Q \to \{1, 2\} \).

**Definition 4.1.5.** A *recurrence* winning condition for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a function \( \pi : Q \to \{0, 1\} \).

**Definition 4.1.6.** A *reactive(1)* winning condition for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a function \( \pi : Q \to \{1, 2, 3\} \).

For simplicity, the following definition describes a reactive(\( n \)) game through a conjunctive parity condition.

**Definition 4.1.7.** A *reactive(\( n \))* winning condition for a parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \pi) \) is a set of parity condition \( \{\pi_1, \pi_2, \ldots, \pi_n\} \) where each parity condition is a function \( \pi : Q \to \{1, 2, 3\} \).
In Figure 2.4, one observes that languages denoted as safety, guarantee, or obligation are all subset of persistence languages. The following lemma shows that given an automaton with a parity condition which is either safety, guarantee, or obligation then it can be replaced with a persistence parity condition without changing the language accepted by the automaton.

**Lemma 4.1.8.** Given a parity game \( G = \langle \Sigma, Q, Q_0, Q_1, \delta, \pi \rangle \) where \( \pi : Q \rightarrow \{0, 1\} \) is an obligation parity condition, let \( \pi' : Q \rightarrow \{1, 2\} \) be the persistence winning condition such that for every state \( q \in Q \)
\[ \pi'(q) = 2 - \pi(q). \]
Then the game \( G' = \langle \Sigma, Q, Q_0, Q_1, \delta, \pi' \rangle \) is equivalent to \( G \).

**Proof.** The lack of cycles of mixed parity in \( G \) implies that \( \pi \) is equivalent (with respect to \( G \)) to \( \pi' \). \( \square \)

Since safety and guarantee properties are subsets of obligation properties, Lemma 4.1.8 can be used to change the parity condition of either safety or guarantee game to that of a persistence game. The usefulness of this lemma becomes apparent when Theorem 4.2.1 is discussed in the next section.

### 4.2 Games with Conjunctive or Disjunctive Parity Conditions

This section deals with a specification that is a disjunction or conjunction of \( \omega \)-regular properties. A deterministic parity automaton is obtained from each property, and then these automata are composed with each other. When the specification is a conjunction (disjunction) of \( \omega \)-regular properties, then the acceptance condition of this composed automaton is defined by a conjunctive (disjunctive) parity condition.

In a conjunctive (disjunctive) parity game the objective of the protagonist is to find a winning strategy that always produces a winning play with respect to all (some) parity conditions. The two types of games are dual of each other; therefore, the focus of the discussion will be on the conjunctive parity games.

Let the specification \( S = P_1 \land P_2 \land \ldots \land P_k \) be a conjunction of properties; the product game \( G = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle \) where each \( \pi \in \{\pi_1, \pi_2, \ldots, \pi_k\} \) corresponds to some property \( P \in S \). Let \( \{\pi_1, \pi_2, \ldots, \pi_n\} \) where \( n \leq k \) be the set of safety winning conditions; this set of safety conditions can be replaced by a single safety condition \( \pi \) such that \( \pi(q) = \prod_{1 \leq i \leq k} \pi_i(q) \). Furthermore, the conjunctive parity winning condition can be converted to an equivalent one thanks to the following observation.
Theorem 4.2.1. Let $\Pi = \{\pi_1, \ldots, \pi_k\}$ ($k > 1$) be a conjunctive winning condition for parity game $G = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle$. Suppose that the largest odd priority in the co-domain of $\pi_k$ is $m$ and the largest odd priority in the co-domain of $\pi_{k-1}$ is $l$. Then $\Pi$ is equivalent to $\Pi' = \{\pi_1, \pi_2, \ldots, \pi_{k-1}, \pi_k\}$ with respect to $G$, where, for each $q \in Q$,

$$
\pi'_{k-1}(q) = \begin{cases} 
    l + 1 & \text{if } \pi_k(q) = m + 1 \\
    \pi_{k-1}(q) & \text{otherwise}
\end{cases}.
$$

Proof. Let $A_p^k$ be the states of $G$ with priority $p$ with respect to $\pi_i$. Suppose a play $\rho$ is winning according to $\pi_k$ and $\pi_{k-1}$ then

$$
\inf(\rho) \cap A_{m+1}^k = \emptyset \text{ and } \inf(\rho) \cap A_{l+1}^{k-1} = \emptyset.
$$

From the definition of $\pi'_{k-1}$ the following holds:

$$
\forall q \in \inf(\rho). \pi_{k-1}(q) = \pi'_{k-1}(q)
$$

implying that

$$
\max (\pi_{k-1}(\inf(\rho))) = \max (\pi'_{k-1}(\inf(\rho))).
$$

Hence $\max (\pi'_{k-1}(\inf(\rho)))$ is odd. Therefore $\rho$ is winning according to $\pi'_{k-1}$ (and $\pi_k$).

Conversely, a play $\rho'$ winning according to $\pi'_{k-1}$ and $\pi_k$ is such that

$$
\inf(\rho') \cap A_{m+1}^k = \emptyset \text{ and } \inf(\rho') \cap A_{l+1}^{k-1} = \emptyset,
$$

where $A_{l+1}^{k-1}$ is the set of states of $G$ with priority $l + 1$ with respect to $\pi'_{k-1}$. From the definition of $\pi'_{k-1}$ the following holds:

$$
\forall q \in \inf(\rho'). \pi'_{k-1}(q) = \pi_{k-1}(q).
$$

Hence $\max (\pi_{k-1}(\inf(\rho'))) = \max (\pi_{k-1}(\inf(\rho)))$ is odd. Therefore $\rho'$ is winning according to $\pi_{k-1}$ (and $\pi_k$).

Notice that the theorem also holds when $m$ is the largest priority in $\pi_k$ but nothing happens in this case. Note that for any state $q$ that satisfies $\forall A_{m+1}^k, \pi_k(q)$ can be set to $m + 1$. In other words, when the token is guaranteed to be in some state $A_{m+1}^k$ after being in some state $q$ then the priority of $q$ can be changed to $m + 1$. This fact is illustrated in the following example.
Example 4.2.2. Consider the games $G = \langle \Sigma, Q_0, Q_1, \delta, \pi \rangle$ and $G' = \langle \Sigma, Q_0, Q_1, \delta, \pi' \rangle$ shown in Figure 4.1. These two games are equivalent because all the paths from $q_1$ lead to states of color 2.

Corollary 4.2.3. Let $\Pi = \{\pi_1, \ldots, \pi_k\}$ ($k > 1$) be a disjunctive winning condition for parity game $G = \langle \Sigma, Q, D, \delta, \Pi \rangle$. Suppose that the largest even priority in the co-domain of $\pi_k$ is $m$ and the largest even priority in the co-domain of $\pi_{k-1}$ is $l$. Then $\Pi$ is equivalent to $\Pi' = \{\pi_1, \pi_2, \ldots, \pi_{k-1}', \pi_k\}$ with respect to $G$, where, for each $q \in Q$,

$$\pi_{k-1}'(q) = \begin{cases} l + 1 & \text{if } \pi_k(q) = m + 1 \\ \pi_{k-1}(q) & \text{otherwise} \end{cases}.$$ 

Proof. The negations of $\Pi$ and $\Pi'$ (discussed in Section 2.3) are conjunctive parity conditions to which Theorem 4.2.1 applies. Since they are equivalent for the antagonist, $\Pi$ and $\Pi'$ are equivalent for the protagonist.

Corollary 4.2.4. Let $\Pi = \{\pi_1, \ldots, \pi_k\}$ ($k > 1$) be a conjunctive winning condition for parity game $G = \langle \Sigma, Q, D, \delta, \Pi \rangle$. Suppose that $\pi_k$ is a persistence winning condition and that the largest odd priority in the co-domain of $\pi_{k-1}$ is $l$. Then $\Pi$ is equivalent to $\Pi' = \{\pi_1, \ldots, \pi_{k-1}'\}$ with respect to $G$, where, for
each \( q \in Q \),
\[
\pi'_{k-1}(q) = \begin{cases} 
  l + 1 & \text{if } \pi_k(q) = 2 \\
  \pi_{k-1}(q) & \text{otherwise} 
\end{cases}.
\]

**Proof.** A play winning for \( \{\pi_1, \pi_2, \ldots, \pi'_{k-1}, \pi_k\} \) is obviously also winning for \( \Pi' \). Conversely, suppose \( \rho' \) is winning according to \( \pi'_{k-1} \) then
\[
\inf(\rho') \cap A^{k-1}_{l+1} = \emptyset.
\]
Since
\[
A^k_2 \cup A^{k-1}_{l+1} = A^{k-1}_{l+1}
\]
the following holds:
\[
\inf(\rho') \cap A^k_2 = \emptyset.
\]
Thus \( \rho' \) is winning according to \( \pi_k \). Therefore \( \Pi' \) is equivalent to \( \{\pi_1, \pi_2, \ldots, \pi'_{k-1}, \pi_k\} \), which, by Theorem 4.2.1, is equivalent to \( \Pi \).

**Corollary 4.2.5.** Let \( \Pi = \{\pi_1, \ldots, \pi_k\} \) \((k > 1)\) be a disjunctive winning condition for parity game \( G = \langle \Sigma, Q, D, \delta, \Pi \rangle \). Suppose that \( \pi_k \) is a recurrence winning condition and that the largest even priority in the co-domain of \( \pi_{k-1} \) is \( l \). Then \( \Pi \) is equivalent to \( \Pi' = \{\pi_1, \ldots, \pi'_{k-1}\} \) with respect to \( G \), where, for each \( q \in Q \),
\[
\pi'_{k-1}(q) = \begin{cases} 
  l + 1 & \text{if } \pi_k(q) = 1 \\
  \pi_{k-1}(q) & \text{otherwise} 
\end{cases}.
\]

**Proof.** The proof directly follows from the proof of Corollaries 4.2.3 and 4.2.4.

**Example 4.2.6.** As an example, consider the LTL games defined by a graph \( \Gamma \), which encodes permissible moves, and an objective \( \varphi \land \psi \), where \( \varphi \) is one of
\[
\varphi_1 = G(\text{favorable} \rightarrow F \text{win})
\]
\[
\varphi_2 = G(\text{favorable} \rightarrow G \neg \text{lose})
\]
\[
\varphi_3 = \text{favorable} \rightarrow (\text{favorable} U \text{win}),
\]

and $\psi$ is either

$$\psi_1 = F (\text{win} \lor \text{lose})$$

or

$$\psi_2 = F G (\text{win} \lor \text{lose})$$.

In these LTL formulae, favorable, win, and lose are atomic propositions labeling the states of $\Gamma$. The initial translation of $\phi \land \psi$ determinizes the NBWs for $\phi$ and $\psi$ and composes them to produce a DPW with conjunctive acceptance condition $\Pi = \{ \pi_\phi, \pi_\psi \}$. According to Figure 2.4, $\varphi_1$ is a recurrence ($G_2$) property, $\varphi_2$ is a safety ($F_1$) property, $\varphi_3$ is an obligation ($F_2 \cap G_2$) property, $\psi_1$ is a guarantee ($G_1$) property, and $\psi_2$ is a persistence ($F_2$) property. In all six games, $\Pi$ can be reduced to an equivalent non-generalized acceptance condition with at most three priorities.

Consider the game $\varphi = \varphi_2$ and $\psi = \psi_2$. Then the games for $\varphi$, $\psi$ and $\varphi \land \psi$ are shown in Figure 4.2 along with the non-generalized game for $\varphi \land \psi$.

Repeated application of Lemma 4.1.8 and Corollary 4.2.4 eliminates all safety, guarantee, obligation, and persistence winning conditions in a conjunctive parity game (except one if there are no other winning conditions) with a maximum increase of one priority in one of the surviving winning conditions (if all remaining winning conditions have an odd priority as maximum priority). Repeated application of Theorem 4.2.1 may further reduce the number of used priorities (hence possibly the index) of one or more parity winning conditions. The “classical” algorithm of [CHP07] runs in

$$O(m \cdot n^{2d}) \cdot \binom{d}{d_1, d_2, \ldots, d_k},$$

(4.1)

where $n = |Q|$, $m = |E|$, $\pi_i : Q \to [d_i + 1]$ is the $i$-th component of the winning condition, $k$ is the number of components, and $d = \sum_{1 \leq i \leq k} d_i$. The simplification of the conjunctive condition afforded by Corollary 4.2.1 and Theorem 4.2.4 improves the bound by reducing $k$ and, in most cases, also $d$.

When the specification is a conjunction of properties, the approach shown in Figure 4.3 is adopted. Theorem 4.2.1 and Corollary 4.2.4 are invoked in the procedure SIMPLIFY; all the strategies for persistent winning conditions\(^1\) are accounted (not just the memoryless ones) without introducing memory. In

\(^1\) All the protagonist's winning states $W$ for a safety winning condition are assigned the priority 1. The protagonist wins the safety game as long as it never visits any state not in $W$.\n
Figure 4.2: $G_\varphi$ where $\varphi = G(f \rightarrow Fw)$, $G_\psi$ where $\psi = GF(w \lor l)$, $G_{\varphi \land \psi} = G_\varphi \times G_\psi$ with conjunctive parity condition, $G'_{\varphi \land \psi} = G_\varphi \times G_\psi$ with non-generalized parity condition.

Line 2.4, the environment’s winning states and winning strategies are collected; the main idea in conjunctive parity game is that a state which is winning for the environment in a game w.r.t. some parity condition $\pi_i$ then it is still a winning state for the environment in the game with conjunctive parity winning condition ($\{\pi_1, \pi_2, \ldots, \pi_1, \ldots, \pi_k\}$ (in such a game the environment’s objective is the negation of the conjunctive parity condition); and it wins the conjunctive parity game by playing the strategy for the parity condition $\pi_i$. In Line 2.5, the composed game is simplified by invoking the various lemmas discussed earlier in this section. Once all the games have been composed with each other and the conjunctive parity condition has been
algorithm CONJUNCTIVEGAME(\{G^1, G^2, \ldots, G^k\})
for each i = 1, 2, \ldots, k
2.1 if i = 1 then G ← G^1
else G ← G × G^i
2.2 let \( \pi \) be the last component of \( \Pi \)
2.3 \((U_0, U_1, T_0, T_1) ← ZIELONKA(G[\Pi ← \pi])\)
2.4 if i = 1 then U ← U_0 and T ← T_0
else U ← U \cup U_0 and T ← T \cup T_0
2.5 G ← SIMPLIFY(G, U_1)
end
3 if \(|\Pi| = 0\) then
7 return(\(\emptyset, Q, \emptyset, \delta\))
4 if \(|\Pi| = 1\) then
8 \((U_0, U_1, T_0, T_1) ← ZIELONKA(G)\)
5 \((U_0, U_1, T_0, S, G) ← ITERATIVECONJGAME(G)\)
6 \((U_0, U_1, T_0, S, G) ← CONJPARITYWIN(G)\)
9 return (U, U_1, T, S)
10 return (U, U_1, T, S)

Figure 4.3: An incremental algorithm for Conjunctive Parity Games. \(S : M × (U_1 \cap Q_1) \rightarrow (U_1 \cap Q_0)\) is a strategy, where \(M\) is the finite memory.

simplified to an equivalent conjunctive parity condition \(\Pi'\), the conjunctive parity condition is analyzed; there is nothing to be done when \(\Pi' = \emptyset\), otherwise if \(\Pi'\) is a singleton then the non-generalized parity game is solved through Zielonka’s algorithm. If \(\Pi' = \{\pi'_1, \pi'_2, \ldots, \pi'_m\}\) where \(m \leq k\) then Zielonka’s algorithm is used to remove states which are losing with respect to some \(\pi'_i \in \Pi'\) until no such state(s) can be removed; this is carried out in the ITERATIVECONJGAME procedure shown in Figure 4.4. When \(\Pi'\) describes a Generalized Streett(1) game (defined in Section 2.4) then the game is converted to a Semi-Generalized Streett(1) condition (defined in Section 4.3.1). The winning strategies for this game can be computed through the procedure of ITERATIVECONJGAME. The details of how the winning strategies for a Semi-Generalized Streett(1) game are computed through the ITERATIVECONJGAME procedure are also discussed in Section 4.3.1. Otherwise, in the final step a conjunctive parity game is played through the procedure of CONJPARITYWIN described in Figure 3.10.
I
TERATIVE
CONJGAME(G)

Q_c ← Q  U ← ∅  T ← ∅  τ ← ∅

do

3.1  Q ← Q_c

3.2  for each i ∈ {1, . . . , k}

3.1.1  (W_0, Q_c, T_0, T_1^i) ← ZIELONKA((Q_c, Q_c ∩ Q_0, Q_c ∩ Q_1, δ, π_i))

3.1.2  U ← U ∪ W_0  T ← T ∪ T_0

3.1.3  G ← SIMPLIFY(G, Q_c)

done

while (Q_c ̸= Q)

4  if REC(Π) or STREETT1(Π) then

    T_1 ← CREATESTRATEGY(T_1^1, T_1^2, . . . , T_1^k)

5  return (U, T_0, Q_c, T_1, (Q_c, Q_c ∩ Q_0, Q_c ∩ Q_1, δ, Π))

Figure 4.4: Iterative algorithm to remove protagonist’s easy-to-identify losing states. The procedure CREATESTRATEGY is an implementation of (4.5).

4.3 Assumptions and Guarantees

In practice, it is often the case that the environment must satisfy its constraints as well; therefore, the specification is realizable if there exists a strategy such that in every play consistent with that strategy the environment constraints are falsified or the system satisfies its constraints. The specification of the environment is provided as a conjunction of properties called assumptions; let the assumptions be the union of a set of safety properties S_e and a set of progress properties L_e. The properties in the specification of the system are called guarantees; let the guarantees be the union of a set of safety properties S_s and a set of progress properties L_s. The specification S states that the assumptions imply the guarantees:

\[ S_e \land L_e \rightarrow S_s \land L_s \]  \hspace{1cm} (4.2)

This specification can be rewritten as following:

\[ \neg S_e \lor \neg L_e \lor (S_s \land L_s) \]  \hspace{1cm} (4.3)

The following section focuses on how the hierarchical approach handles the synthesis of S in an incremental manner. The discussion is split in two cases, when S is a GR(1) specification (described in Section 2.6) and when S is not a GR(1) specification.
4.3.1 GR(1) Specifications

Specifications of the form of GR(1) are particularly interesting because it has been shown that many reactive systems in practice can be described by a GR(1) specification. In this section, it is shown how a game obtained from a GR(1) specification is converted to a conjunctive parity game. This parity game can be solved without employing Chatterjee’s algorithm (discussed in Section 3.9.2) for solving games with conjunctive parity condition.

Since the specification $S$ in the form of Equation 4.2 or 4.3 is neither a conjunction nor a disjunction of properties, the game $G_S$ obtained by composing the parity games corresponding to every property in $S$ is a game with generalized parity condition. The algorithm of Figure 4.3 cannot be used directly to solve $G_S$. As discussed in Section 2.4 one can obtain a new game $G'_S$ from $G_S$ such that the acceptance condition is a Streett condition with only one pair (or a non-generalized parity condition). However, this conversion requires two counters to be added in $G_S$, which significantly increases the size of $G'_S$, especially when the set of recurrence properties in $L_s$ is large. (It shall be seen later in this section that the counter to convert the recurrence properties in $L_e$ to a single recurrence property is still needed.) To obtain a smaller game from $G_S$ one relies on Lemmas 4.3.1 and 4.3.5, which are special cases of Corollary 4.2.5.

**Lemma 4.3.1.** Let $\Pi = \{\pi_1, \ldots, \pi_k\}$ ($k > 1$) be a disjunctive winning condition for a parity game $G = \langle Q, Q_0, Q_1, \delta, \Pi \rangle$. Suppose that $\pi_{k-1}$ is a persistence winning condition and $\pi_k$ is a recurrence winning condition; then $\Pi$ is equivalent to $\Pi' = \{\pi_1, \ldots, \pi'_{k-1}\}$ with respect to $G$, where, for each $q \in Q$,

$$\pi'_{k-1}(q) = \begin{cases} 3 & \text{if } \pi_k(q) = 1 \\ \pi_{k-1}(q) & \text{otherwise} \end{cases}.$$

**Proof.** The proof follows directly from Corollary 4.2.5. \hfill $\square$

The disjunction of a recurrence and a persistence condition unsurprisingly results in a Streett winning condition with one pair. Consider a specification of the form of $S$ where both $L_e$ and $L_s$ contain one recurrence property each denoted by $\pi_e$ and $\pi_s$ respectively. Given $G_S = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle$, where $\Pi$ is a disjunction between $\neg \pi_e$ and $\pi_s$ (the negation of a recurrence property is a persistence property,) one can
obtain $G'_S = \langle \Sigma, Q, Q_0, Q_1, \delta, \pi' \rangle$, where $\pi'$ is a Streett condition with a single pair; thanks to Lemma 4.3.1 the games $G_S$ and $G'_S$ are equivalent (notice that in this particular case no counters were added to $G'_S$). The following lemma is a generalization of Lemma 4.3.1.

**Lemma 4.3.2.** Let $G = \langle Q, Q_0, Q_1, \delta, \Pi \rangle$ be a generalized parity game. Let $\pi$ be a persistence condition and let $\Pi^c = \{\pi^c_1, \ldots, \pi^c_k\}$ be a conjunctive winning condition such that every $\pi^c_i \in \Pi^c$ is a recurrence condition. Let $\Pi$ be the disjunction of $\pi$ and $\Pi^c$. Then $\Pi$ is equivalent to the conjunctive condition $\hat{\Pi} = \{\hat{\pi}^c_1, \ldots, \hat{\pi}^c_k\}$ with respect to $G$, where, for each $q \in Q$ and $0 < i \leq k$,

$$
\hat{\pi}^c_i(q) = \begin{cases} 
3 & \text{if } \pi^c_i(q) = 1 \\
\pi(q) & \text{otherwise}.
\end{cases}
$$

**Proof.** The winning condition $\Pi$ is equivalent to the conjunctive winning condition $\Pi' = \{\pi'_1, \ldots, \pi'_k\}$ where $\pi'_i = \{\pi, \pi^c_i\}$ is a disjunctive winning condition. By Lemma 4.3.1 $\pi'_i$ is equivalent to $\hat{\pi}^c_i$. Therefore $\Pi^c$ is equivalent to $\Pi'$ and hence to $\Pi$. \qed

Notice that in $\hat{\Pi}$

$$
\forall 0 < i \leq k : \forall 0 < j \leq k . A^i_1 \cap A^j_2 = \emptyset \tag{4.4}
$$

because all the $A^i_1$ and $A^j_2$ are derived from the same persistence condition $\pi$. A conjunctive parity winning condition that satisfies (4.4) is referred to as a Semi-Generalized Streett(1) condition; therefore a GR(1) specification can be converted to a game with a Semi-Generalized Streett(1) condition. The exact details of how a GR(1) specification is converted to a game with a Semi-Generalized Streett(1) condition are discussed after the procedure of Figure 4.6 is discussed. However, the basic idea is that the recurrence conditions of $L_e$ are converted to a single recurrence condition $\pi_e$ by adding a counter in $G_S$ (the details have been described in Section 2.4). Then the disjunction between the negation of $\pi_e$ and the recurrence conditions of $L_e$ can be reduced to a Semi-Generalized Streett(1) game as proven in Lemma 4.3.2.

It was discussed in Section 3.3 that a special method of solving $G_S$ was proposed in [PPS06]. The method being proposed here is to convert $G_S$ to $G'_S$ which is a Semi-Generalized Streett(1) game. It is now shown that the procedure CONJPARITYWIN for solving conjunctive parity games is not used to solve
Theorem 4.3.3 shows that the winning region of a game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \Pi) \) with a Semi-Generalized Streett(1) condition \( \Pi \) can be computed through repeated application of Zielonka’s algorithm (which cannot be directly used to solve a game with a conjunctive parity condition).

Let \( G = (Q, Q_0, Q_1, \delta, \Pi) \) be a parity game where \( \Pi \) is either a conjunction of recurrence conditions or it can be converted to a Semi-Generalized Streett(1) condition. Let \( Q_c \subseteq Q \) be the set of states obtained through the sequential procedure of Figure 4.4. Let the set \( S = \{s_1, \ldots, s_k\} \) be the memory; then the strategy \( \tau_\Pi : S \times (Q_c \cap Q_1) \rightarrow S \times Q_c \) is obtained from the set of strategies \( \tau = \{\tau_1, \ldots, \tau_k\} \) as follows:

\[
\tau_\Pi((s_i, q)) = \begin{cases} 
(s_i, \tau_i(q)) & q \in ((Q_c \setminus A^i_3) \cap Q_1), \\
(s_j, \tau_j(q)) & q \in (Q_c \cap A^i_3 \cap Q_1) \text{ and } j = 1 + (i \mod k).
\end{cases}
\] (4.5)

The set \( S \) denotes the state of a counter that resets to 1 after counting from 1 to \( k \).

**Theorem 4.3.3.** For the parity game \( G = (\Sigma, Q, Q_0, Q_1, \delta, \Pi) \), where \( \Pi \) is a Semi-Generalized Streett(1) condition, the strategy \( \tau_\Pi \) is a winning strategy for the conjunctive player.

**Proof.** Suppose the conjunctive player employs the strategy \( \tau_\Pi \) to play game \( G \). Then irrespective of the environment’s strategy, the states appearing in the run \( \rho \) of \( G \) are contained in \( Q_c \). This is because every sub-strategy of \( \tau_\Pi \) keeps the game within \( Q_c \). Suppose \( \rho \) is such that all sub-strategies of \( \tau_\Pi \) are used infinitely often; in other words, every memory state of \( S \) is visited infinitely often in \( \rho \). From the definition of \( \tau_\Pi \) it is observed that \( \tau_\Pi \) switches from its sub-strategy \( \tau_i \) only when the run \( \rho \) reaches a state in \( A^i_3 \cap Q_c \); this implies that \( \forall 0 < i \leq k . \inf(\rho) \cap A^i_3 \neq \emptyset \). Hence, \( \rho \) is a winning run.

Suppose \( \rho \) is produced by \( \tau_\Pi \) when it only switches between its sub-strategies a finite number of times. Then the run \( \rho \) gets stuck in some memory state \( s_i \in S \) and there exists a sub-strategy \( \tau_i \) that generates an infinite suffix of \( \rho \) (once the memory gets stuck in state \( s_i \)). By definition of \( \tau_i \), the run \( \rho \) is

---

2 The co-domain of every recurrence is set to \( \{2, 3\} \), where every state assigned priority 1 is assigned priority 3 and every state assigned priority 0 is assigned the priority 2. This avoids case-splitting in the definition of \( \tau_\Pi \).

3 Recall that non-generalized parity winning conditions admit positional strategies.
winning with respect to the condition $\pi_i$, therefore, $\inf(\rho) \subseteq A_i^1$. From Equation (4.4) it is concluded that $\forall 0 < i \leq k \cdot \inf(\rho) \cap A_i^2 = \emptyset$. Therefore, $\rho$ is winning. Hence $\tau_{\Pi}$ is a winning strategy of $G$.

The application of a procedure derived from Theorem 4.3.3 is now outlined. When the specification $S$ is encountered, the first step is to create a game where the environment has maximum freedom to satisfy its assumptions $S_e \land L_e$, which is illustrated in the following example. (The games shown in examples 4.3.4, 4.3.6, 4.3.7, and 4.4.2 are all input-based games, which are discussed in Chapter 6.)

**Example 4.3.4.** Consider the specification $S = S_e \land L_e \rightarrow S_s \land L_s$ where

$$
\begin{align*}
X_{ed} &= \{a, b\} & X_s &= \{x\} \\
X_{ep} &= \emptyset \\
S_e &= \{G(\neg a \lor \neg b)\} & L_e &= \{L_e^1, L_e^2\} \\
L_e^1 &= GF(a) & L_e^2 &= GF(b) \\
S_s &= \{G(a \land b \rightarrow X((a \land x) \lor (b \land x)))\} & L_s &= \{L_s^1, L_s^2\} \\
L_s^1 &= GF(a \land x) & L_s^2 &= GF(b \land x)
\end{align*}
$$

The games corresponding to the properties in $S_e$ and $L_e$ are shown in Figure 4.5; these games are composed with each other. The resulting game $G_e = G_{S_e} \times G_{L_e^1} \times G_{L_e^2}$ is then restricted to environment’s winning states denoted by $\mathcal{G}_e$, which is also shown in Figure 4.5.

The algorithm of Figure 4.3 is used to compute the environment’s winning states $W$ of the game $G_e = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle$ obtained from the properties in $S_e \cup L_e$. Let $\mathcal{G}_e = \langle \Sigma, W, W \cap Q_0, W \cap Q_1, \delta, \Pi \rangle$ denote the game $G_e$ restricted to the environment’s winning states $W$. The set of states $Q \setminus W$ are winning for the system (because in these states the environment is unable to satisfy its assumptions).

Once $\mathcal{G}_e$ has been obtained its acceptance condition is negated (negation of a parity acceptance condition has been discussed in Section 2.3). The modified algorithm of Figure 4.6 is used to establish the realizability of $\mathcal{S}$. Let the set of games $\{G^1, G^2, \ldots, G^l\}$ correspond to the games from the set $S_s$ and the set of games $\{G^{l+1}, G^{l+2}, \ldots, G^k\}$ correspond to the games from the set $L_s$. These games along with
$G_e$ are passed as arguments to the procedure INCREMENTALGAME of Figure 4.6. The following lemma is employed during this procedure to extend the to first check the realizability of $S_e \land L_e \rightarrow S_s$.

Lemma 4.3.5. Let $\Pi = \{\pi_1, \pi_2\}$ be a disjunctive winning condition for the game $G = (\Sigma, Q, Q_0, Q_1, \delta, \Pi)$. 
Suppose $\pi_1$ is a safety winning condition and $\pi_2$ is an arbitrary winning condition such that the protagonist loses the game $G_2 = \langle \Sigma, Q, Q_0, Q_1, \delta, \pi_2 \rangle$ everywhere. Then the protagonist’s winning states in $G$ are the winning states in the game $G_1 = \langle \Sigma, Q, Q_0, Q_1, \delta, \pi_1 \rangle$.

Proof. Let $Q = A_0^1 \cup A_1^1$, where $A_p^1$ is the set of states assigned priority $p$ by $\pi_1$. In a safety game the states in $A_0^1$ have no transitions to states in $A_1^1$; therefore the set $W$ of winning states in game $G_1$ is a subset of $A_1^1$.

Since the antagonist wins game $G_2$, it has a winning strategy from the states $A_0^1$ in $G$. The set $A_1^1 \setminus W$ is winning for the antagonist in $G_1$, because from these states, the antagonist has a strategy to force a visit of $A_0^1$. Therefore, the antagonist has a winning strategy for the states $Q \setminus W$ in $G$; which is to force a visit of states in $A_0^1$ and then switch to the strategy for $G_2$. \hfill \Box

Without the assumption on $\pi_1$ the lemma does not hold because the strategies that win $\pi_2$ may lose $\pi_1$. If, however, $\pi_1$ is a safety condition, then the antagonist’s winning strategy for $\pi_2$ in $A_1^1$ keeps the play within $A_0^1$. It can easily be shown that this lemma holds even if $\pi_2$ represents a generalized parity condition.

In the modified algorithm shown in Figure 4.6, Lemma 4.3.5 is applied on Line 2.2. For instance, a new game $G = \langle \Sigma, Q, Q_1, Q_2, \delta, \Pi \rangle$ with $G = G_e \times G_1$ is obtained, where the winning condition is $\Pi = \Pi_e \lor \Pi_1$ (the game $G_1$ is a safety game). It is known from the construction of $G_e$ that the antagonist wins $G_e$; therefore, the antagonist wins the game $\langle \Sigma, Q, Q_0, \delta, \Pi_e \rangle$. From Lemma 4.3.5 one can compute the winning states of $G$ by playing the game $\langle \Sigma, Q, Q_0, Q_1, \delta, \pi_1 \rangle$. This process is repeated until all the safety properties of $S_S$ have been composed and the resulting game is restricted to system’s winning states.

The following example is an illustration of this concept.

**Example 4.3.6.** This example is a continuation of Example 4.3.4. Given the games $G_e$ with a conjunctive parity condition $\Pi_e$ and $G_{S_S}$ (shown in Figure 4.7) with a safety condition $\pi_s$, the game $G_s = G_e \times G_{S_S}$ is generated with a generalized parity condition $\Pi_s = \{\neg \Pi_e, \pi_s\}$ (this parity condition represents the specification $S_e \rightarrow S_s$). One can observe that the game $G_{S_S}$ is losing for the system but the game $G_s$ (in this case $G_s = G_S$) is winning for the system. The winning states of the game $G_s$ can be computed from the game $G_s[\Pi_s \leftarrow \pi_s]$ (because of Lemma 4.3.5).

Once all the games obtained from the individual properties of the specification have been composed,
algorithm INCREMENTALGAME(\{G_e, G^1, G^2, \ldots, G^l, G^{l+1}, \ldots, G^k\})

1.1 \quad G \leftarrow G_e

2 \quad \textbf{for each} i = 1, 2, \ldots, k

2.1 \quad G \leftarrow G \times G^i

2.2 \quad \textbf{if} i \leq l \textbf{then}

2.2.1 \quad (U_0, U_1, T_0, T_1) \leftarrow ZIELONKA(G[\Pi \leftarrow \pi_i])

2.2.2 \quad \textbf{if} i = 1 \textbf{then}

2.2.2.1 \quad U \leftarrow U_0 \textbf{ and } T \leftarrow T_0

2.2.2.2 \quad \textbf{else}

2.2.2.3 \quad U \leftarrow U \cup U_0 \textbf{ and } T \leftarrow T \cup T_0

2.2.3 \quad G \leftarrow SIMPLIFY(G, U_1)

3 \quad \textbf{if} |\Pi| = 0 \textbf{then}

3.1 \quad \textbf{return}(\emptyset, Q, \emptyset, \delta)

4 \quad \textbf{if} \textsc{Rec}(\Pi_e) \textbf{ and } \textsc{Rec}(\Pi) \textbf{ then}

4.1 \quad G \leftarrow \textsc{CREATESEMIGENSTREETT}G(E)(G)

4.2 \quad (U_0, U_1, T_0, S, G) \leftarrow \textsc{ITERATIVECONJGAME}(G)

4.3 \quad \textbf{return}(U \cup U_0, U_1, T \cup T_0, T_1)

5 \quad G \leftarrow \textsc{CREATEDISJUNCTIVEGAME}(G)

8 \quad (U_0, U_1, S, T_1) \leftarrow \textsc{DISJPARITYWIN}(G)

9 \quad U \leftarrow U \cup U_0 \textbf{ and } T \leftarrow T \cup S

10 \quad \textbf{return}(U, U_1, T, T_1)

Figure 4.6: An incremental algorithm for Conjunctive Parity Games. \(S : M \times (U_1 \cap Q_1) \rightarrow (U_1 \cap Q_0)\) is a strategy, where \(M\) is the finite memory.

two set of parity conditions are left, the first set is \(\Pi_e\) obtained from \(G_e\), which represents the assumptions, while the set \(\Pi\) represents the guarantees from the set \(L_s\). If both \(\Pi_e\) and \(\Pi\) only contain recurrence conditions then the game \(G\) can be converted to a Streett(1) game with just one Streett pair as discussed in Section 2.4. The generalized parity game \(G = \langle \Sigma, Q, Q_0, Q_1, \delta, \{\Pi_e, \Pi\} \rangle\) is converted to generalized parity game \(G' = \langle \Sigma, Q', Q'_0, Q'_1, \delta', \{\{\pi_e\}, \Pi\} \rangle\) with the parity condition \(\pi_e\) as shown in Section 2.4 (at this point \(\Pi_e\) is a disjunction of persistence properties which is first converted to a conjunction of recurrence properties \(\neg\Pi_e\) and then converted to a single recurrence property \(\neg\pi_e\) and finally it is negated to obtain a single persistence property \(\pi_e\)). Recall that \(\Pi\) is a conjunction of recurrence properties and therefore Lemma 4.3.2 is invoked to simplify \(\{\{\pi_e\}, \Pi\}\) to obtain a Semi-Generalized Streett(1) condition. Finally, procedure \textsc{ITERATIVECONJGAME} invokes Theorem 4.3.3 and solves the Semi-Generalized Streett(1) game.

The following example illustrate the final two steps of the procedure.

Example 4.3.7. This example is a continuation of Example 4.3.6. Given the game \(G_s\), \(G_{L^1_s}\) and \(G_{L^2_s}\) the
Figure 4.7: (a) The game $G_{S_s}$ and the game $G_e$ are composed with each other; after playing the composed game, one finds the winning strategies for the specification $S_e \land L_e \rightarrow S_s$ when the environment does not violate the antecedent ($S_e \land L_e$). (b) The game $G_s$ which is obtained from $G_e \times G_e$ (the conjunctive parity condition of $G_e$ has been negated) by removing the system’s losing states (the unreachable states are marked with yellow color). (c) Notice that the system cannot win $G_s$, however it can win $G_e \times G_s$. (d) In later examples, the unreachable states of $G_s$ will not be shown and the last component of the generalized parity condition is redundant. The coloring function in this game is an example of a generalized parity condition where the inner parentheses imply a conjunction while the outer parentheses imply a disjunction.
Figure 4.8: In the generalized parity game $G = G_s \times G_{L_1} \times G_{L_2}$ every state in $G$ is connected to every other state (the unreachable states of $G$ are not shown). The states have been marked with the label of the incoming transitions (in this particular game every incoming transition of a state has the same label, which is simply a coincidence). The generalized priority condition of every state is shown under the state.

$A = \neg a \land \neg b$
$B = \neg a \land b \land \neg x$
$C = \neg a \land b \land x$
$D = a \land \neg b \land \neg x$
$E = a \land \neg b \land x$

The game $G = G_s \times G_{L_1} \times G_{L_2}$ is generated. The game shown in Figure 4.8 is a game with a generalized parity condition.

One can solve the game $G$ using the $\mu$-calculus formula of (3.2). However, one can obtain a game
$G'$ shown in Figure 4.10 by composing $G$ with the counter shown in Figure 4.9.

Finally, the acceptance condition of $G'$ is simplified by invoking Lemma 4.3.2. The resulting game is shown in Figure 4.11.

The next section discusses the synthesis of the specification $S$ when it is not a GR(1) specification.

4.3.2 General Case

When either of the set of progress properties $L_e$ or $L_a$ contains a persistence or a reactive property then the specification is not a GR(1) specification; therefore, it cannot be converted to a generalized Streett(1) game. One can always obtain a game with a non-generalized parity condition from it (see Section 2.4. However, one would like to avoid the addition of memory which causes the size of the game to increase dramatically. Therefore, when the specification $S_e \land L_e \rightarrow S_s \land L_s$ is not a GR(1) specification then one can obtain a generalized parity game $G = \langle \Sigma, Q, Q_0, Q_1, \delta, \{\Pi_e, \Pi\} \rangle$ where $\Pi_e$ is a disjunctive parity condition while $\Pi$ is a conjunctive parity condition. In the next step, the conjunctive parity condition $\Pi$ is converted to a non-generalized parity condition $\pi$ through the introduction of index appearance record as briefly discussed in Section 2.4). The new generalized parity game $G' = \langle \Sigma, Q', Q'_0, Q_1, \delta', \Pi' \rangle$ where $\Pi' = \{\Pi_e, \{\pi\}\}$ is a disjunctive parity game (because $\Pi_e$ is a disjunction of parity conditions and $\Pi_e \vee \pi$
Figure 4.10: The game $G'$ obtained by composing the counter in Figure 4.9 with $G$ (the unreachable states of $G'$ are not shown). Instead of having the transitions drawn, these have been listed. For instance the state $A_0$ implies that the token is in state $A$ of $G$ and 0th state of the counter; this state has transitions to states $A_0, B_1, C_1, D_0, E_0$. The label of the transition from $A_0 \rightarrow B_1$ is $B$ (which represents $\neg a \land b \land \neg x$ as listed in Figure 4.8).

is still a disjunction of parity conditions). One may be able to invoke Corollary 4.2.3 to simplify the parity condition $\Pi'$. Finally, the game is solved using the algorithm CONJPARITYWIN; however in $G'$ the system is the disjunctive player while the environment is the conjunctive player.

Unlike the case when $S$ is a GR(1) specification, a simple counter may not be sufficient to convert $L_e$ to a non-generalized condition. Furthermore, when $S$ is not a GR(1) specification then INCREMENTAL-GAME falls back on Chatterjee’s algorithm for conjunctive parity conditions.
Figure 4.11: The conjunctive parity game $\tilde{G}$ is obtained from $G'$ shown in Figure 4.10 by invoking Lemma 4.3.2.

4.4 General Reactive($n$) Specification

A specification which satisfies the syntactic characterization of a GR($n$) formula (described in Section 2.6) is of the form described by the following equation:

$$\bigwedge_{1 \leq k \leq m} \left[ \bigwedge_{1 \leq i \leq p_k} (G F K_k^i) \right] \rightarrow \bigwedge_{1 \leq i \leq n_k} (G F J_k^i)$$  \hspace{1cm} (4.6)

where $J_k^i$ and $K_k^i$ are past-time LTL formulae and $m, p_k$ and $n_k$ are natural numbers. Let $S$ be the GR($n$) specification, then it can be observed from (4.6) that $S = S_1 \wedge S_2 \wedge \cdots \wedge S_n$ is a conjunction of $n$ specifications, where every $S_i \in S$ is a GR(1) specification.

An efficient way to convert the specification $S$ to a game is to obtain $n$ different games, one for each GR(1) specification $S_i$ as discussed in Section 4.3.1. These $n$ games can then be composed with each other. If no counters are added then the composed game is a game with generalized parity condition and, as stated earlier, there is no known algorithm to solve such a game. On the other hand, one can add two
counters to each game to convert the GR(1) condition to a Streett condition with a single pair. In this case the composed game will have at most $n$ Streett pairs and such a game can then be solved using Chatterjee’s algorithm for conjunctive parity games. However, adding two counters for each game causes the size of the game to increase dramatically. One can also add a single counter in each game and then obtain a Semi-Generalized Streett(1) condition by invoking Lemma 4.3.2 as shown in Section 4.3.1. (The counter converts the recurrence properties in the antecedent to a single recurrence property.) The composed game is a Streett game. (Alternatively, one could have converted the consequent to a single recurrence property through a counter. However, in this case the game for $S_1$ results in a disjunctive parity game and the game for $S$ is a game with a negation of a generalized parity condition.)

As it has been stated in Section 2.3, a Streett game can be interpreted as a conjunctive parity game $G = (\Sigma, Q, Q_0, Q_1, \delta, \Pi)$, where $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$ and the co-domain of each parity condition $\pi \in \Pi$ is $\{1, 2, 3\}$. Suppose the conjunctive parity condition $\Pi$ satisfies the following condition

$$\exists 1 \leq i \leq k. \exists 1 \leq j \leq k. \forall q \in Q. (\pi_i(q) = 1 \rightarrow \pi_j(q) \neq 2) \land (\pi_j(q) = 1 \rightarrow \pi_i(q) \neq 2). \tag{4.7}$$

In words, (4.7) states that in $\Pi = \{\pi_1, \pi_2, \ldots, \pi_i, \ldots, \pi_j, \ldots \pi_k\}$ there exist two parity conditions $\pi_i$ and $\pi_j$ such that every state in $Q$ that is assigned priority 2 by $\pi_i$ is not assigned priority 1 by $\pi_j$ and every state in $Q$ that is assigned priority 2 by $\pi_j$ is not assigned priority 1 by $\pi_i$. Notice that the parity condition $\{\pi_i, \pi_j\}$ of $G$ satisfies (4.4). The solution of a Streett game with Streett condition that satisfies (4.7) can be simplified through an observation inspired by the proof of Lemma 4.3.5. Given $G$ when a subgame $G_s$ of $G$ is considered such that this subgame does not contain any states with priority 2 or 3 w.r.t. $\pi_i$ and the odd player cannot force a visit to the states with priority 3 w.r.t. $\pi_i$ from the states in $G_s$ nor can the even player force a visit of states with priority 2 w.r.t. $\pi_i$ from the states in $G_s$ then one observes that $G_s$ does not contain any states with priority 2 w.r.t. $\pi_j$. This observation leads to Lemma 4.4.1 that simplifies the solution of a Streett game $G$. 

Let \( G_i = \langle \Sigma, \tilde{Q}, (\tilde{Q} \cap Q_0), (\tilde{Q} \cap Q_1), \delta, \Pi \rangle \) be a subgame of \( G \) where

\[
Q' = Q \setminus \text{attr}_1(G, A_i^3)
\]

\[
G' = \langle \Sigma, Q', (Q' \cap Q_0), (Q' \cap Q_1), \delta, \Pi \rangle
\]

\[
\tilde{Q} = Q' \setminus \text{attr}_2(G', (A_i^1 \cap Q'))
\]

then the following lemma describes how the Streett condition of \( G_i \) can be simplified.

**Lemma 4.4.1.** Let \( \Pi = \{ \pi_1, \pi_2, \ldots, \pi_i, \ldots, \pi_j, \ldots, \pi_k \} \) be a Streett condition for a parity game \( G = \langle \Sigma, Q, Q_0, Q_1, \delta, \Pi \rangle \) such that (4.7) holds, then the conjunctive player’s winning states of \( G_i = \langle \Sigma, \tilde{Q}, (\tilde{Q} \cap Q_0), (\tilde{Q} \cap Q_1), \delta, \Pi \rangle \) are the winning states of \( G'_i = \langle \Sigma, \tilde{Q}, (\tilde{Q} \cap Q_0), (\tilde{Q} \cap Q_1), \delta, (\Pi \setminus \{ \pi_i, \pi_j \}) \rangle \).

**Proof.** The proof follows the fact that \( \forall q \in \tilde{Q} \cdot \pi_i(q) = 1 \) and \( \forall q \in \tilde{Q} \cdot \pi_j(q) = 1 \lor \pi_j(q) = 3 \). Therefore, every winning strategy of the conjunctive player in the game \( G'_i \) produces runs that only visit states with priority 1 w.r.t. \( \pi_i \) and priority 1 or 3 w.r.t. \( \pi_j \). Hence, a winning strategy in \( G'_i \) is a winning strategy in \( G_i \). \( \square \)

Without invoking Theorem 4.3.3 and Lemma 4.4.1 whenever possible during the solution of Streett games Chatterjee’s algorithm for conjunctive parity games generates a strategy that requires more memory than desired. This fact can be observed in the following example, where one winning strategy has been generated by invoking Lemma 4.4.1 and another winning strategy has been generated without invoking it.

**Example 4.4.2.** Consider the game \( \tilde{G} \) shown in Figure 4.10. All the states of the game \( \tilde{G} \) are winning for the conjunctive player. The two conditions \( \pi_1 \) and \( \pi_2 \) satisfy (4.7). The states of the subgame \( \tilde{G}_1 \) are \( \{ A_0, D_0, A_1, B_1, C_1 \} \) (because \( \text{attr}_1(\tilde{G}, A_i^1) = A_3^1 \) where \( A_3^1 = \{ E_0, E_2 \} \), while \( \text{attr}_0(G', A_i^2) = A_2^1 \) where \( A_2^1 = \{ D_2 \} \).

When the Lemma 4.4.1 is invoked in the subgame \( \tilde{G}_1 \) then the conjunctive player wins by default. However, Chatterjee’s algorithm assumes that it still a single bit of memory to fulfil the obligations of \( \pi_1 \) and \( \pi_2 \) in \( \tilde{G}_1 \) because it has no knowledge that there are no states with priority 0 or 2 in \( \tilde{G}_1 \).

One can invoke the procedure of Carton and Maceiras to simplify the acceptance condition of \( \tilde{G}_1 \) but
this procedure will only simplify $\pi_j$ and assign priority 1 to all the states of $\tilde{G}_1$. Chatterjee’s algorithm is not aware that no memory is required to win $\tilde{G}_1$ as there are no priority 0 states w.r.t. $\pi_1$ or $\pi_2$.

Coming back the game $\tilde{G}$, when Lemma 4.4.1 is invoked then a memory $S = \{s_1, s_2\}$ is required, and the strategy function is as follows:

- **If the memory is in state $s_1$ and the game is in $\{A_0, D_0, B_1, C_1, D_2\}$ and the environment plays $(a \land \neg b)$ then the system asserts $x$ and changes memory to $s_2$; otherwise the system is not obligated to assert or deassert $x$.**

- **If the memory is in state $s_2$ and the game is in $\{A_0, D_0, E_0, A_1, B_1\}$ and the environment plays $(\neg a \land b)$ then the system asserts $x$ and changes memory to $s_1$; otherwise the system is not obligated to assert or deassert $x$.**

(The states $\{(s_1, E_0), (s_1, E_2), (s_2, C_1)\}$ are unreachable while the system does not need a particular strategy in states $\{(s_2, D_2), (s_2, E_2)\}$ as these states are transient.) This is also the strategy computed when Theorem 4.3.3 is invoked.

On the other hand, the strategy function generated by Chatterjee’s algorithm (when the procedure of Carton and Maceiras simplifies the subgames) without invoking Lemma 4.4.1 requires the memory $S = \{(s_1, t_1), (s_1, t_2), (s_2, t_1), (s_2, t_2)\}$. This strategy function is as follows:

- **If the memory is in state $(s_1, \{t_1, t_2\})$ and the game is in $\{A_0, D_0, B_1, C_1, D_2\}$ and the environment plays $(a \land \neg b)$ then the system asserts $x$ and changes the memory to $(s_2, t_1)$; otherwise after each turn the memory changes from $(s_1, t_1)$ to $(s_1, t_2)$ or $(s_1, t_2)$ to $(s_1, t_1)$.**

- **If the memory is in state $(s_2, \{t_1, t_2\})$ and the game is in $\{A_0, D_0, E_0, A_1, B_1\}$ and the environment plays $(\neg a \land b)$ then the system asserts $x$ and changes the memory to $(s_1, t_1)$; otherwise after each turn the memory changes from $(s_2, t_1)$ to $(s_2, t_2)$ or $(s_2, t_2)$ to $(s_2, t_1)$.**

It has already been explained in Example 3.9.6 that when the objective in the game $\tilde{G}$ (such as visiting a state with priority 3 w.r.t. $\pi_1$ when memory is $s_1$) is fulfilled, the memory for the subgames is reset.
The import of Lemma 4.4.1 is that the solution of a game obtained from a \( GR(n) \) specification can be significantly simplified when the game is obtained through the composition of \( n \) Streett(1) games as discussed earlier in this section.

### 4.5 Well-Separated Environment

When a GR(1) specification \( S \) of the form (4.2) is encountered, the approach of [PPS06] reduces it to the following form:

\[
(S_e \rightarrow S_s) \land (L_e \rightarrow L_s).
\]  

This is convenient because the safety properties in the specification can be efficiently dealt with before the progress properties. However, in [KP11], the authors proved that this reduction is only correct when the environment is well-separated; otherwise, the realizability of the specification reduced to the form of Equation 4.3 through the solution of a game between the system and the environment may produce a false negative by suggesting that the specification is not realizable when in fact it is. The following example from [KP11] illustrates this fact:

**Example 4.5.1.** Consider the specification \( S \) defined by \( S_e \land L_e \rightarrow S_s \land L_s \), where

\[
\begin{align*}
S_e &= \text{G} \neg X e \\
L_e &= \text{G} \text{F} e \leftrightarrow s \\
S_s &= \text{G} X (e \leftrightarrow s) \\
L_s &= \text{G} \text{F} s \quad \text{and} \\
X_{ed} &= \{ e \} \\
X_s &= \{ s \} \\
X_{ep} &= \emptyset.
\end{align*}
\]

When the system sets \( s = \top \), the environment is unable to satisfy \( S_e \land L_e \) against this strategy; therefore, the antecedent is falsified. Hence, \( S \) is realizable. On the other hand, if \( S \) is reduced to \( (S_e \rightarrow S_s) \land (L_e \rightarrow L_s) \), the system loses its ability to falsify \( L_e \), therefore, the specification \( S \) is incorrectly proven to be unrealizable.

An environment that relies on the cooperation of the system to satisfy its liveness constraints is a non well-separated environment. For specifications with non well-separated environments, the rewriting may
incorrectly prove the specification to be unrealizable. The solution proposed in [KP11] is to first detect if
the specification describes a well-separated environment or not. This is done by checking if the specification
\[ E = \neg(S_e \land L_e) \] is realizable. If it is not realizable then the environment is well-separated and it is safe to
use the reduction. On the other hand, if \( E \) is realizable then one should use the following reduction:

\[
(S_s \mathcal{W} S_e) \land (S_e \land L_e \rightarrow L_s).
\] (4.9)

In practice, when the environment is non well-separated, the above solution is not a viable option
because \( S_s \mathcal{W} S_e \) has to be treated as a monolithic property. The approach of [Ehl10] handles it efficiently by
adding an extra signal to remember if the environment has violated its constraints or not. In the hierarchical
approach, an extra signal is not required because the specification is never reduced to (4.8). Therefore, one
can handle both cases without any performance compromises. It has been described in Section 4.3.1 that a
game \( G_e \) is created where the environment must be able to satisfy \( S_e \land L_e \), while the system is free to play
with any strategy. If the system has a winning strategy in \( G_e \) then the system has a winning strategy for the
specification \( S \). However, if the system does not have a winning strategy in \( G_e \) then \( G_e \) is obtained (the game
\( G_e \) is restricted to environment’s winning states). The system must now satisfy its obligations represented
by \( S_s \land L_s \) in \( G_e \). The details of this final step have already been discussed in Section 4.3.1.

In this chapter, the specification \( S \) is converted to a game by exploiting the hierarchy of \( \omega \)-regular
properties (discussed in Section 2.6). This often results in a game which has fewer states and its acceptance
condition is simpler. This results in significant runtime improvements in solving the game.
Chapter 5

Transition Constraints

In the game-based approach to synthesis, the specification is converted to a game between the system and the environment. A deterministic implementation is obtained from the game graph and a system’s winning strategy. However, there are obstacles to extract an efficient hardware implementation. On the one hand, a large space must be explored to find a strategy that has a concise representation. On the other hand, the transition structure inherited from the game graph may correspond to a state encoding that is far from optimal. In practice, the specification of a reactive system often contains a large number of safety properties; this is exploited in the hierarchy based approach discussed in Chapter 4 to improve the representation of the game and simplify its solution. In this chapter, an improvement to the hierarchy based synthesis framework is proposed through which one can obtain significantly smaller implementations from such specifications.

The amount of sequential logic (for instance the number of state-holding elements such as flip-flops) in an implementation is directly dependent on the transition function of the game, while the amount of combinational logic is dependent primarily on the system’s winning strategy and to some extent the transition function. The authors of [BGJ+07a] describe a heuristic to select a winning strategy that attempts to minimize the amount of combinational logic in the implementation. Similarly the authors of [EKH12] describe a heuristic to select a winning strategy that attempts to minimize the amount of sequential logic. However, none of the current approaches address the issue of converting the specification to a more compact game that can lead to improved run-times and the quality of results (QoR).

A specification consisting of a set of \( \omega \)-regular properties can be easily converted to a single \( \omega \)-regular property by conjoining all the properties in the set. However, the conversion of this property to a
deterministic game is computationally very expensive because of the state explosion problem. In the tools based on the approaches of [SS12, EKH12] property is converted to a deterministic automaton and then these automata are composed with each other. A deterministic game is the derived from the resulting automaton. The tool based on the approach of [BGJ+07b] requires the decomposition of the property automata into simpler properties (transition-constraints defined later in this chapter) manually. The transition function of the game is then inferred from these simple properties automatically. The drawback of these approaches is that the state space of the composed game may contain significant amount of redundancies (e.g. bisimilar states, unreachable states). Furthermore, when the symbolic representations of the automata are composed with each other, the symbolic representation of the resulting automaton may be extremely inefficient. The reasons of inefficiency in the symbolic representation of the resulting automaton are discussed in Section 5.1. Remedying these problems may not be a simple task.

Converting the specification to a game which does not contain any any redundancies is not a realistic goal. On one hand, identifying bisimilar states in an game obtained from a progress property is computationally expensive. On the other hand, symbolic algorithms (specifically BDD based algorithms) struggle on games that are very compact, such games are obtained after removing all the redundancies and then optimally encoding them. The approach described in this chapter provides a general framework through which the game obtained from the specification has fewer bisimilar and unreachable states while guaranteeing the existence of an efficient encoding of the game. It focuses on the safety properties of the specification. The winning conditions of either player with respect to the safety properties are first computed in an abstract manner which is very efficient. Then these winning conditions are expressed as a parameterized transition function. This parameterized transition is then composed with the games obtained from the remaining properties as done in [SS12].

The approach discussed in this chapter was motivated by the fact that any safety property describes a language that can be expressed as a relation between current and next state values of the input and output signals (where input signals are controlled by the environment and the output signals are controlled by the system). When the conjunction of all the safety properties describe a language that can be generated by a relation the problem of sequential synthesis is converted to a problem of combination synthesis. Other-
wise, just enough memory is added so that the conversion of sequential synthesis problem to combinational synthesis problem is possible. The sections 5.2, 5.4, 5.5, and 5.7 discuss the core details of this approach. The combinational synthesis problem is the extraction of a parameterized transition function for the system player which guarantees the satisfaction of all the safety properties in the specification while capturing the freedom of system. The combinational synthesis is formulated as a set of Boolean equations described in Section 5.8. The general solutions of these Boolean equations capture all the possible ways in which a player can satisfy the safety properties in the specification. An additional advantage of this approach is that it is symbolic and thus adept at manipulating a large set of safety properties.

The parameterized representation of the general solutions of the Boolean equations is the reason why this approach can be used with the incremental synthesis framework of Chapter 4. A simple retiming heuristic is also discussed in Section 5.12. This allows the symbolic representation of the parameterized solution to be further optimized. As a result, both the sequential and combinational logic needed by the hardware implementation is often reduced by an order of magnitude. This retiming heuristic is not applicable when the framework discussed in this chapter is not employed.

In [Mad11], the author makes the claim that when synthesizing imperative programs from an $\omega$-regular specification, one should not synthesize a transition system and then convert it to an imperative. One of the main challenges is that the quality (for instance when this transition system is implemented in hardware then quality implies the amount of combinational and sequential logic required) of the transition system is sensitive to the syntax of the specification. The author then provides an alternative way based on the Safraless approach (discussed in Section 3.10) to generate the imperative program directly that satisfies the specification but is less sensitive to the syntax of the specification. The framework based on the relation-based later discussed in this chapter provides an alternative through which the transition system synthesized from the specification is less sensitive to the syntax of the specification.

5.1 Challenges in the Encoding of Transition Structures

Symbolic algorithms have resulted in significant gains in the field of model checking and automatic synthesis. The focus of this section are the challenges in making the symbolic representation of an automa-
ton or a game more efficient. The symbolic representations are sensitive to the encoding function and the sets being represented symbolically. The approach of converting each property of the specification to a deterministic automaton and then composing the automata leads to a very inefficient encoding of the resulting automaton. In Chapter 8, the results obtained from using SAFETY-FIRST approach while employing the automata based approach to convert the specification to a game demonstrates the inefficiency of this conversion technique. The inefficiency in the symbolic representation of the game makes the task of finding a better implementation (the one that uses less combinational and sequential logic) significantly more difficult.

It has already been discussed in Section 2.8 that choosing a good symbolic encoding of an arbitrary sized set is extremely hard (the quality of the encoding varies with the details of the function defined over this domain). The relation-based approach discussed in this chapter uses fewer symbolic variables to encode the game obtained from the safety properties of the specification. This results in improvement in the efficiency of symbolic algorithms used to solve the games. An additional advantage is that when BDD-based symbolic algorithms are used, the use of fewer symbolic variables often simplifies the search for a good variable order.

In this section, the reasons for the inefficiencies in the symbolic encoding are discussed, specially the case when the symbolic representation of two automata are composed. The presentation of these issues is restricted to safety automata because in practice the inefficient handling of safety automata is the primary reason why the game obtained from the specification contains significant amount of redundancies. On top of that, minimizing automata obtained from progress properties is not a straightforward task.

Let $A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle$ be an $\omega$-automaton. The language of a state $q \in Q$ denoted by $L(q)$ is defined as the set of words that produce a run that visits the state $q$. A reduced automaton $A_r = \langle \Sigma, Q_r, q_{in}, \delta_r, \pi_r \rangle$ is obtained from $A$ such that $Q_r = \{ q \in Q \mid L(q) \neq \emptyset \}$ and for all $q \in Q_r$ the transition function $\delta_r(q, \sigma) = \delta(q, \sigma) \cap Q_r$ and the parity function $\pi_r(q) = \pi(q)$. Both automata $A$ and $A_r$ accept the same language because a word in $L(A)$ still has an accepting run in $A_r$. In the reduced safety automata the parity condition is implicit because all states in $Q_r$ have priority 1. One recalls that in a safety automaton, the states are either colored 1 or 0 and from every 0 colored state there is no transition to a 1 colored state. No word accepted by a safety automaton visits any 0 colored state of that automaton, therefore a reduced automaton can only have 1 colored states. Hence every infinite run of a reduced safety automaton
is an accepting run. Reduced automata are often incomplete in the sense that when the token is in certain state not all letter of the alphabet have a corresponding outgoing transition.

A reduced safety automaton $A_r$ is **irredundant** if it does not contain language equivalent states. The notion of irredundancy is only defined for deterministic safety automata because the notion of language equivalence cannot be used to minimize a non-safety $\omega$-automaton. For instance in a Büchi automaton $A = (\{\neg p, p\}, \{q_1, q_2\}, q_1, \delta, \{q_2\})$ shown in Figure 5.1, both states are language equivalent but $A$ cannot be further minimized. However, when the automaton is a deterministic safety automaton then the notion of direct simulation\(^1\) is equivalent to language equivalence. The following lemma shows that in a reduced deterministic safety automaton two language equivalent states direct simulate each other.

**Lemma 5.1.1.** Given a reduced deterministic safety automaton $A = (\Sigma, Q, q_{\text{in}}, \delta, \pi)$ such that $q_1 \in Q$ and $q_2 \in Q$ are language equivalent then $q_1 \preceq_d q_2$ and $q_2 \preceq_d q_1$.

**Proof.** Suppose $q_1$ is trying to direct simulate $q_2$ then if $q_2$ selects the next letter that takes it to $q_2' \in Q$ then $q_1$ has a strategy to go to another state $q_1' \in Q$ where both $q_1'$ and $q_2'$ are language equivalent and by definition of reduced automaton these states are colored 1.

Suppose $q_2$ wants to jump ship and selects a letter that is not readable by $q_2$ then this letter is not readable by $q_1$ either (a letter that is not readable by a state implies that it has no outgoing transition when that letter is read). This is because this letter extends the prefix such that it is not in the language. If this letter was readable by $q_1$ and led to a state whose language was not empty then $q_1$ and $q_2$ were not language equivalent.

\(^1\) Fair and delayed simulation are weaker notions than direct simulation while being stronger notions than language equivalence. Delayed simulation has been defined in [EWS01].
equivalent. If after reading this letter $q_1$ was led to a state with an empty language then the definition of reduced automaton is violated.

The state $q_1$ can match every move of $q_2$ such that the priorities of the target states are also matched, therefore, $q_1$ direct simulates $q_2$. With the same reasoning it can be shown that $q_2$ simulates $q_1$. \qed

The Lemma 5.1.1 shows that one can simplify the automaton such that no two states direct simulate each other, such a deterministic safety automaton is irredundant.

5.1.1 State Encodings of Automata and Games

This thesis is primarily concerned with improving the efficiency of symbolic algorithms for solving games. Therefore, it is important to understand the core issues that arise when the specification is converted to a game. It has already been discussed that the specification is converted to a game by obtaining a symbolic game for each property and then composing all the games. Given an automaton $A = \langle (X_{ed} \cup X_s \cup X_{ep}, Q, q_0, \delta, \pi) \rangle$, the input alphabet is $2^{X_{ed}} \times 2^{X_s} \times 2^{X_{ep}}$ while boolean state variables are needed to encode the state space of this automaton (it can also be interpreted as a game where the environment controls $X_{ed} \cup X_{ep}$ and system controls $X_s$). The next value of each state variable is a function of inputs and/or state variables. Cyclic dependencies between state variables, that is the next value of variable $s_i$ depends on the current value of $s_j$ and the next value of $s_j$ depends on the current value of $s_i$ imply that the transition structure is more tightly connected. When the state variables of a transition structure have significant amount of such kind of dependencies, one cannot increase the efficiency of image and pre-image computations through some clever use of quantification\(^2\) schedules. Cyclic dependencies between state variables may be eliminated by choosing an alternate state encoding, however this is not the solution discussed in this chapter. Consider the automata shown in Figure 5.2. If $A_\Phi$ is encoded with two state variables

\(2\) Given the transition function $\delta$ in the form of a relation $\Delta(S, X, S')$ between current state variables $S$, next state variables $S'$ and inputs $X = X_{ed} \cup X_s \cup X_{ep}$ then the pre-image of a set of states $T(S')$ in an automaton is given by

$$\exists X. \exists S'. \Delta(S, X, S') \land T(S') .$$

If one wants to compute the attractor with respect to the system of $T(S')$ then

$$\forall X_{ed}. \exists X_s \forall X_{ep}. \exists S'. \Delta(S, X, S') \land T(S') .$$
$s_1$ and $s_2$ then $s_1$ depends on $s_2$.

$$s_1 = (a \land b \land c) \lor (a \land \neg b \land \neg c) \lor (\neg c \land s_2),$$

$$s_2 = (a \land b) \lor (\neg c \land s_2).$$

On the other hand if $A_\Phi$ is encoded with three state variables $s_0$, $s_1$ and $s_2$ then there is no dependency between any of these variables.

$$s_0 = (a \land b) \lor (\neg c \land s_0),$$

$$s_1 = a \land b \land \neg c,$$

$$s_2 = a \land \neg b \land \neg c.$$

Similarly, in Figure 5.3 if $A_\Phi$ is encoded with two state variables $s_1$ and $s_2$ then there is a cyclic dependency between these two state variables.

$$s_1 = a \lor (\neg s_2 \land b),$$

$$s_2 = (\neg a \land b) \lor (a \land \neg b \land \neg s_2) \lor (a \land c \land s_1) \lor (a \land \neg c \land s_1).$$

On the other hand, if $A_\Phi$ is encoded with three state variables $s_0$, $s_1$ and $s_2$ then there is no cyclic dependency between any of these variables and $s_0$ and $s_1$ do not depend on any of $s_0$, $s_1$ or $s_2$.

$$s_0 = b,$$

$$s_1 = a,$$

$$s_2 = (a \land c \land s_2) \lor (a \land \neg b \land \neg s_1) \lor (a \land b \land \neg c \land s_1).$$

If given an automaton $A$ encoded with $n$ state variables, finding a non-cyclic or sparse encoding with more state variables is not a straightforward task. However, the conversion of specification to a game results in a sparse encoding and one may re-encode when it is beneficial and avoid re-encoding when the benefits to be had are minimal which has been discussed in the remainder of this sub-section.

Preliminary investigation in finding efficient re-encoding did not prove to be fruitful. Given a redundant encoding of an automaton, the value of a state variable can often be computed as a function of
Figure 5.2: (a) $\Phi_1 = G((a \land b) \rightarrow X(c \land \neg a))$ (b) $\Phi_2 = G((a \land \neg c) \rightarrow (b \leftrightarrow Xb))$ (c) $\Phi = \Phi_1 \land \Phi_2$. There are two unreachable states in $A_\Phi = A_{\Phi_1} \times A_{\Phi_2}$ denoted by the set $U$, while the set $X$ denotes the don’t care states. When $A_\Phi$ is obtained from $\Phi$ directly then the encoding is shown as the labels of the states.
other state variables. One can remove the variable by substituting the dependant variable by its function. This simplifies the encoding in the sense that the number of symbolic variables has decreased. However, the observation was made that removing a dependant variable \( v \) that is functionally dependent on a set of state variables \( U \) such that \( |U| > 2 \), has an adverse effect on the performance of the BDD operations while computing the solution to the synthesis game. This is primarily due to increase in dependency between the remaining state variables, (e.g., if \( k \) depends on \( v \) then by removing \( v \) one has made \( k \) dependent on \( U \)). (One may look at the impact of removing a variable \( v \) by measuring the increase of the supports of the variables that depend of \( v \), but this was not considered in the preliminary investigation.) The task of finding an efficient re-encoding of the game proved to be an extremely ambitious goal.

The relation based approach of obtaining a parameterized transition function from safety properties side-steps the issue of re-encoding the state space of the safety game. However, this approach is able to capture the actual intent of the safety properties. Therefore, the encoding of the parameterized transition function is a straightforward task. This encoding may still contain some redundancy in its encoding, however, this has not proven to be a cause of serious concern from the results of the experimental setup.

### 5.1.2 Bisimilar States in Automata and Games

When two automata that contain neither unreachable states nor bisimilar states are composed with each other, the resulting automaton may have unreachable states as shown in Figures 5.2, 5.3 and 5.4. The resulting automaton may also contain simulation equivalent states. It is easy to detect such states in safety automata, therefore the discussion will be restricted to safety automata only. Consider the example in Figure 5.4 where the original automata \( A_{\Phi_1} \) and \( A_{\Phi_2} \) do not contain any simulation equivalent states, but the states \( q_1 \) and \( q_2 \) of \( A_{\Phi} \) are direct simulation equivalent.

The presence of either unreachable or simulation equivalent states is the reason for the symbolic representation of the resulting automaton to use more state variables than necessary. One can invoke the appropriate notion of simulation minimization to remove the simulation equivalent states and simplify the transition structure. This may in turn provide the opportunity to re-encode the automaton. However, the challenges in re-encoding an automaton have been discussed in the previous section. In the particular case
Figure 5.3: There are two unreachable states in $A_\Phi = A_{\Phi_1} \times A_{\Phi_2}$ denoted by the set $U$, while the set $X$ denotes the don’t care states. $A_\Phi$ can be re-encoded with using two state variables as well.
Figure 5.4: (a) $\Phi_1 = G((\neg a \land c) \rightarrow ((a \lor \neg c) \mathcal{R} b))$ (b) $\Phi_2 = G(\neg a \rightarrow X b)$ (c) $\Phi = \Phi_1 \land \Phi_2$. There are two unreachable states in $A_\Phi$ which are not shown.
of Figure 5.4, the removal of simulation equivalent state simplifies the transition structure which leads to a simpler encoding of the resulting automaton shown in Figure 5.5.

$$\begin{align*}
q_0 = s_0 & \quad q_2 = \neg s_0 \land \neg s_1 \\
q_3 = \neg s_0 \land s_1 \\
\overline{s_0} = (s_0 \lor b) \land \neg a \land \neg c \\
\overline{s_1} = a \land \neg b
\end{align*}$$

The approach discussed in this chapter produces a parameterized transition function from the safety properties. This function describes a game graph which contains fewer bisimilar states.

### 5.1.3 Deterministic Automata with Pseudo Inputs

Given a nondeterministic game, one can resolve the nondeterminism by introducing pseudo inputs [Cho74]. This makes the game deterministic. The introduction of pseudo inputs to resolve the nondeterminism of the game can sometimes be used to simplify the symbolic representation of the game. However, this can only be done if the pseudo input can exclusively be controlled by one of the players. In other words, one has to evaluate if the control of a pseudo is assigned to a certain player then does it give this player more freedom then allowed. Consider the game $G$ of Example 3.5.5, the determinization of $A_\varphi$ through the addi-
tion of pseudo inputs does not solve the problem. Because if either player gets the exclusive control of this input the player gets more freedom and the winning strategy cannot be used to synthesize an implementation for such a game.

The following example describes a situation where one can introduce pseudo inputs and simplify the symbolic representation of a game.

**Example 5.1.2.** Consider the property \( \varphi_1 = G((-a \to (b \leftrightarrow X b)) \land (a \lor c)) \). The automaton \( A \) shown in Figure 5.6 accepts the language defined by \( \varphi_1 \). In \( A \) the input alphabet \( \Sigma_A \) is the set of minterms \( \mathcal{A}(I_A) \), where \( I_A = \{a, b, c\} \).

The automaton \( A' \) has an extra input \( b_i \) and its input alphabet \( \Sigma_{A'} \) is the set of minterms \( \mathcal{A}(I_{A'}) \), where \( I_{A'} = \{a, b, c, b_i\} \). Whenever the current value of \( a = \top \) then the current value of the pseudo input \( b_i \) controls the next value of \( b \), otherwise the value of \( b \) is determined by the state of \( A' \). The projection of \( L(A') \) on \( \Sigma_A^\omega \) is \( L(A) \).

Given a specification \( \Phi = \varphi_1 \land \varphi_2 \), where \( \varphi_2 = GF(b \leftrightarrow c) \), \( X_{ed} = \{c\} \), \( X_{ep} = \emptyset \), and \( X_s = \{a, b\} \), in the input-based game derived from \( A' \) the system cannot satisfy the property \( \varphi_2 \) even when the system has the ability to select the next value of \( b \) through \( b_i \). (The variable \( b \) is a dependant variable in \( A' \) and its value depends on the state of the automaton. The transitions in \( A' \) are controlled by the independent variables \( a \) and \( b_i \). Even if the system always asserts \( a \) the environment can always deassert \( c \) in \( s_1 \) and assert \( c \) in \( s_0 \).) This is because the environment can always set the value of \( c = \neg b \) after the system has chosen the next value of \( b \) through \( a \) and \( b_i \) by staying in the same state or moving into the other state. Hence, the recurrence condition of \( \varphi_2 \) can never be satisfied unless the environment cooperates. Therefore, the game derived from \( A' \) can not be used to check the realizability of \( \Phi \).

However, in the input-based game derived from \( A \) the system can always select the appropriate value of \( b \) after knowing what the environment has assigned to \( c \). The system can always assert \( a \) and stay in the state \( q_q \) and can always set the value of \( b = c \). With this strategy, the system has a strategy to satisfy \( \varphi_2 \) in the game derived from \( A \). Since \( A \) satisfies \( \varphi_1 \) the system has a strategy to satisfy \( \Phi \).

In the case when \( X_{ed} = \emptyset \) and \( X_{ep} = \{c\} \) then both input-based games derived from \( A \) and \( A' \)

---

\(^3\mathcal{A}(I_A)\) is defined in (2.12).
with $\varphi_2$ as the winning condition can be used to check the realizability of $\Phi$ as both games prove that $\Phi$ is unrealizable with the given partition of variables in $X$. (The property $\Phi$ is defined over the variables in $X$, where $X = X_{ed} \cup X_s \cup X_{ep}$.)

The automaton $A$ requires seven symbolic variables (three for the inputs and four for current and next states), while the automaton $A'$ requires six symbolic variables (four for inputs and two for current and next states). Since the search for a good variable order is often expensive, fewer symbolic variables imply a smaller search space. Therefore fewer symbolic variables contribute to the efficiency of the synthesis algorithm. $A'$ is used in place of $A$ to generate the two-player game wherever its use produces correct results.

This example gives an insight into one of the ways in which a game can be optimized. Even though pseudo inputs can resolve non-determinism, it is not easy to establish the ownership of these pseudo inputs. However, in the context of solution of Boolean equations discussed in Section 5.8, pseudo inputs (parameters) are used to resolve the nondeterminism between various solutions of the Boolean equations, which are then used to generate a deterministic automaton as shown in Example 5.1.2. It is discussed in Section 5.12 how to optimize $A$ to obtain $A'$ and the conditions under which such an optimization can be used to check the realizability of $\Phi$.

This section has discussed some of the important reasons that cause the inefficient conversion of the specification to a game. In the remaining sections of this chapter, a new conversion approach is proposed. This approach converts safety properties of the specification to a significantly more compact game which does not stress the symbolic algorithms in finding an implementation that guarantees the satisfaction of the specification. Even though the new conversion approach does not address any of these problems directly, the impact of some of the problems discussed here is greatly minimized when the new approach is employed.

### 5.2 $\mathcal{R}$-Generability and 1-Definiteness

This section characterizes the subset of the safety languages that can be generated by a relation. These languages are denoted as $\mathcal{R}$-generable languages in [Eme90]. The linguistic view of a subset of $\omega$-regular
Figure 5.6: $\varphi_1 = G((\neg a \rightarrow (b \leftrightarrow X b)) \land (a \lor c))$. 
languages described in [Eme90] is presented in this section. Then the equivalent characterization of these languages in terms of \( \omega \)-automata that accept them is also presented along with a procedure to detect these languages through the automata that accept them. In this chapter, if a property that describes an \( \mathcal{R} \)-generable language satisfies certain syntactic rules then such a property is denoted as a \textbf{transition constraint}. Such a property directly describes the transition structure of an automaton that accepts the language specified by it. An efficient synthesis procedure from transition constraints is described later in this chapter. Therefore, it is practically relevant to detect \( \mathcal{R} \)-generable languages and be able to use the more efficient synthesis procedure.

\textbf{Definition 5.2.1.} A set of infinite words \( W \subseteq \Sigma^\omega \) is \textbf{\( \mathcal{R} \)-generable} if there exists a binary relation \( \mathcal{R} \) on \( \Sigma \) such that a sequence \( w_0w_1w_2\ldots \) is in \( W \) iff \( \forall i \geq 0, (w_i, w_{i+1}) \) is in \( \mathcal{R} \).

The language generated by \( \mathcal{R} \subseteq \Sigma \times \Sigma \) is denoted by \( L(\mathcal{R}) \subseteq \Sigma^\omega \). It has been shown in [Eme90] that a set of infinite words \( W \subseteq \Sigma^\omega \) is \( \mathcal{R} \)-generable iff it is suffix-closed, fusion-closed, and limit-closed\(^4\). The definitions of these three concepts are as follows:

\textbf{Definition 5.2.2.} A set of infinite words \( W \subseteq \Sigma^\omega \) is \textbf{suffix-closed} if for every word \( w_0w_1w_2\ldots \in W \) then the suffix \( w_1w_2\ldots \in W \).

\textbf{Definition 5.2.3.} A set of infinite words \( W \subseteq \Sigma^\omega \) is \textbf{fusion-closed} if whenever word \( xvy \in W \) and \( avb \in W \), then \( xvb \in W \) (and \( avy \in W \)).

\textbf{Definition 5.2.4.} A set of infinite words \( W \subseteq \Sigma^\omega \) is \textbf{limit-closed} if whenever the words \( w_0a, w_0w_1b, w_0w_1w_2c, \ldots \) belong to \( W \), then the limit of the prefixes \( w_0, w_0w_1, w_0w_1w_2, \ldots \), which is the infinite word \( w_0w_1w_2\ldots \) is also in \( W \).

Suffix-closed languages exist that are neither limit-closed nor \( \omega \)-regular. Let \( \Sigma = \{a, b\} \) and consider the following language:

\[
\bigcup_{n>0} \left( \bigcup_{0 \leq i \leq n} b^i(a^n b^n)^\omega \cup \bigcup_{0<i<n} a^i b^n (a^n b^n)^\omega \right).
\]

\(^4\) In the context of synthesis, it is convenient to drop the requirement that the relation be total. As part of the realizability check of a specification, a subset of the alphabet is computed over which the relation is indeed total.
This language is neither safety nor an \( \omega \)-regular language, but it is a suffix-closed language. For instance, given \( n = 2 \) any suffix of the word \( abb(aabb)\omega \) is included in the language; however, this is not an \( \omega \)-regular language because this language requires an automaton which can count to \( n \) where \( n \) is unbounded. Similarly, this is not a safety language because the infinite word \( a^\omega \) is not in the language; however any finite prefix of this word can be extended such that it belongs to the above language. As another example, the LTL property \( G F p \) on alphabet \( \Sigma = \{p, \neg p\} \) defines a language that is \( \omega \)-regular, suffix-closed, and fusion-closed, but not limit-closed. (The deterministic \( \mathcal{B} \)uchi automaton is shown in Figure 5.1.)

The \( \omega \)-automata that can recognize suffix-closed, fusion-closed and limit-closed languages have special characteristics. These characteristics are now studied as these allow one to identify these languages by examining the transition structure of the automata accepting them. It was shown in [Lan69] that the \( \omega \)-regular languages that are limit-closed are accepted by safety automata. It is now shown that an \( \omega \)-regular language that is suffix-closed is accepted by some \( \omega \)-automaton that is initially free.

**Definition 5.2.5.** An automaton \( A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle \) is **initially free** iff

\[
\forall \sigma \in \Sigma . \forall q \in Q . \forall q' \in Q . q' \in \delta(q, \sigma) \rightarrow q' \in \delta(q_{in}, \sigma).
\]

That is, in an initially-free automaton, if there is a transition from \( q \) to \( q' \) labeled \( \sigma \), then there is also a transition from \( q_{in} \) to \( q' \) with the same label. An example of initially free automaton is shown on the left in Figure 5.7. This automaton is initially free because every letter that appears in a word accepted by the language is accepted by the initial state. Similarly, the irredundant automaton that accepts the language defined by \( G(a \rightarrow b) \) is initially free. (The initial state is only state and that accepts the letters \( \{a \land b, \neg a \land b, \neg a \land \neg b\} \). However, the letter \( \{a \land \neg b\} \) does not appear in any word in the language, that is why it is not read by the initial state.)

**Lemma 5.2.6.** An \( \omega \)-regular language \( W \subseteq \Sigma^\omega \) is suffix-closed iff it is accepted by an initially-free automaton over \( \Sigma \).

**Proof.** Suppose \( W \) is suffix-closed. Since \( W \) is \( \omega \)-regular, it is accepted by an NBW \( A \). Consider \( A' = \langle \Sigma, Q \cup \{q_{in}'\}, q_{in}', \delta', \pi' \rangle \) obtained from \( A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle \) by cloning the initial state so that \( q_{in}' \) has no
incoming transitions, and then adding any transitions required to make it initially free; that is,

\[ \forall \sigma \in \Sigma . \delta'(q'_\text{in}, \sigma) = \{ q' \mid \exists q \in Q . \exists \sigma \in \Sigma . q' \in \delta(q, \sigma) \} \]

\[ \forall q \in Q . \forall \sigma \in \Sigma . \delta'(q, \sigma) = \delta(q, \sigma) \]

\[ \pi'(q'_\text{in}) = \pi(q_\text{in}) \]

\[ \forall q \in Q . \pi'(q) = \pi(q) . \]

The two automata \( A' \) and \( A \) accept the same language. Clearly, cloning the initial state does not change the language. Since adding transitions cannot remove words from the accepted language, \( L(A) \subseteq L(A') \). It remains to prove that \( L(A') \subseteq L(A) \). Let \( w \) be accepted by \( A' \). Every transition of an accepting run is also in \( A \), except for the first one, because the initial state cannot be reached once it has been left. Suppose the initial transition is labeled \( \sigma \) and leads to \( q' \). Then there is a state \( q \) in \( A \) with a transition labeled \( \sigma \) to \( q' \) and \( w \) is also accepted from \( q \). Hence, \( w \) is a suffix of a word \( uw \in W \). Since \( W \) is suffix-closed, \( w \in W \) and is accepted by \( A \).

Suppose \( W \) is accepted by an initially-free automaton \( A \). Then, it is \( \omega \)-regular. If \( W = \emptyset \), it is trivially suffix-closed. Suppose not and consider a word \( uw \in W \). Let \( \rho \) be an accepting run of \( uw \) and let \( q \) and \( q' \) be the states of \( \rho \) reached after reading \( u \) and \( uw_0 \), respectively. By definition of initially-free automaton, \( A \) has a transition from \( q_\text{in} \) to \( q' \) labeled \( w_0 \). Therefore, it accepts \( w \) and hence \( W \) is suffix-closed. \( \square \)

To check whether an \( \omega \)-automaton \( A \) accepts a suffix-closed language, one constructs an initially-free automaton \( A' \) as described in the proof of Theorem 5.2.6. If \( L(A') \subseteq L(A) \) then the language accepted by \( A \) is suffix-closed. When \( A \) is deterministic, if the initial state fair-simulates every other state of the automaton then \( L(A) \) is suffix-closed. In particular, when \( A \) is a safety automaton, if the initial state simulates every other state then \( L(A) \) is suffix-closed.

The notion of definiteness is now described because it provides the connection between fusion-closed \( \omega \)-regular languages and \( \omega \)-automata.

**Definition 5.2.7.** An \( \omega \)-automaton \( A \) is \( \mu \)-**definite** if \( \mu \) is the least integer such that the current state of \( A \) is completely determined by the most recent \( \mu \) letters read.
The value of $\mu$ for a $\mu$-definite automaton can be determined through various methods. One of these methods is through synchronizing trees, which is briefly described here. A full discussion can be found in [Koh78]. A tree is constructed whose root is labeled with the set of states of the automaton, while the remaining nodes of the tree are labeled with subsets of states. The label of each node defines the current uncertainty on the states and the label of the children is the set of states which can be visited after reading a certain letter. If the automaton is $\mu$-definite, the tree will eventually have a level where all nodes are labeled with singletons. The value of $\mu$ is given by the longest path in the tree. The following result is a special case of the test for definiteness [PRS63, Koh78]:

**Lemma 5.2.8.** An automaton is 1-definite iff for every input letter $\sigma \in \Sigma$, there exists a state $q \in Q$ such that for every state $q' \in Q$, $\delta(q', \sigma)$ is either $\emptyset$ or $q$.

In other words, all transitions enabled by $\sigma$ lead to the same state $q$. A 1-definite automaton must be deterministic because having one state from which two or more states are reachable when reading some letter is directly against the definition of 1-definiteness. This observation extends to $\mu$-definiteness if every state that exhibits nondeterminism can be reached via a path of length at least $\mu - 1$.

The notion of definiteness is relaxed to yield the connection between fusion-closed languages and definite automata.

**Definition 5.2.9.** An automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle$ is half definite iff for every letter $\sigma \in \Sigma$ the states in $\{ q' \mid \exists q \in Q. q' \in \delta(q, \sigma) \}$ are language equivalent.

**Lemma 5.2.10.** If an $\omega$-regular language $W \subseteq \Sigma^\omega$ is fusion-closed, then all deterministic automata that accept it are half-definite. If an $\omega$-regular language $W \subseteq \Sigma^\omega$ is accepted by a half-definite deterministic automaton, then it is fusion-closed.

**Proof.** Suppose $W$ is accepted by a deterministic half-definite $\omega$-automaton $A$. Given words $u_1cw_1$ and $u_2cw_2$ in $W$, suppose that after reading the prefixes $u_1c$ and $u_2c$, $A$ is in states $q_1$ and $q_2$. $A$ must accept both $w_1$ and $w_2$ from both states because $q_1$ and $q_2$ are language equivalent. Therefore, it accepts $u_1cw_2$ and $u_2cw_1$. Hence, the language $W$ is fusion-closed.
Suppose $W$ is fusion-closed. Let $A$ be a deterministic automaton that accepts $W$. Consider two words $u_1cw_1 \in W$ and $u_2cw_2 \in W$ such that $A$ is in state $q_1$ and $q_2$ after reading the prefixes $u_1c$ and $u_2c$ respectively. Since $W$ is fusion-closed then both states accept both $w_1$ and $w_2$. This implies that $q_1$ and $q_2$ are language equivalent. Hence, $A$ is half-definite.

**Corollary 5.2.11.** An $\omega$-regular language $W \subseteq \Sigma^\omega$ is fusion-closed and limit-closed iff it is accepted by a 1-definite safety automaton.

**Proof.** Suppose $W$ is accepted by a 1-definite safety automaton $A$, then from the definition of 1-definiteness, one concludes that $A$ is deterministic. Consider the words $u_1cw_1$ and $u_2cw_2$ in $W$; then the automaton $A$ is in some state $q$ after reading the prefixes $u_1c$ and $u_2c$ because $A$ is 1-definite. The state $q$ accepts the suffixes $w_1$ and $w_2$, therefore, $u_1cw_2$ and $u_2cw_1$ are both in $W$. Hence, $W$ is a fusion-closed language.

Suppose $W$ is a fusion-closed and limit-closed language accepted by a deterministic automaton $A$. By Lemma 5.2.10 $A$ is half-definite. Since $W$ is a safety language the automaton $A$ can be assumed to be a safety automaton. One can obtain an irredundant automaton $A'$ from $A$. The safety automaton $A'$ does not contain any language equivalent states. Therefore, for every word $ucw \in \Sigma$ the automaton $A'$ is in a unique state after reading the prefix $uc$. Hence, $A'$ is a 1-definite automaton.

The following theorem characterizes the $\omega$-regular languages that are $R$-generable in terms of the structure of their accepting automata. This provides an efficient membership test for safety languages that can be generated by relations.

**Theorem 5.2.12.** A language $W \subseteq \Sigma^\omega$ is $R$-generable iff it is accepted by an initially-free, 1-definite safety automaton.

**Proof.** If a set $W \subseteq \Sigma^\omega$ is generated by a relation $R$, an initially-free, 1-definite safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, Q \rangle$ can be built as follows. For each letter $\sigma \in \Sigma$ that appears in some pair of $R$, a state $q_{\sigma}$ is added to $Q$, distinct from $q_{in}$. Let $\delta(q_{\sigma}, \sigma') = q_{\sigma'}$ for each pair $(\sigma, \sigma') \in R$. Moreover, let $\delta(q_{in}, \sigma) = q_{\sigma}$ for every letter $\sigma$ that appears in first position in some pair of $R$. This guarantees that $A$ is initially-free because $q_{in}$ is connected to every state in $Q \setminus q_{in}$ that have at least one outgoing transition. Then, $A$ accepts $W$. 

If $A = \langle \Sigma, Q, q_{in}, \delta, Q \rangle$ is an initially-free, 1-definite safety automaton accepting $W$, a relation $R$ is built as follows. The pair of letters $(\sigma, \sigma')$ is added to $R$ when $\delta(q, \sigma') = q'$ and $\sigma$ is the letter that labels the transitions into $q$. (No pair is added to $R$ for a state with no incoming transitions.) Then, $R$ generates $W$.

It is instructive to prove this result in a different way, which is less constructive, but emphasizes the connection between the linguistic and automata-theoretic viewpoints.

**Alternate proof.** Suppose $W$ is $R$-generable. Then it is suffix-closed, fusion-closed and limit-closed. By Corollary 5.2.11 the language $W$ is accepted by a 1-definite safety automaton. Therefore, there exists a 1-definite safety automaton $A$ that accepts $W$. This $A$ must be initially free. Suppose not. Then there are a letter $\sigma$ and a state $q$ in $A$ such that $\delta(q_{in}, \sigma) = \emptyset$. (It cannot be a state different from $\delta(q, \sigma)$ because $A$ is 1-definite.)

However, a word $w \in W$ can be formed by concatenating a finite word leading to $q, \sigma$, and any word accepted from $\delta(q, \sigma)$. (Such a word exists because the automaton is reduced.) The suffix starting with $\sigma$ is not in $W$, contradicting the suffix-closedness of $W$.

Suppose $A$ is an initially-free, 1-definite safety automaton that accepts $W$. Then Lemma 5.2.6 and Corollary 5.2.11 imply that $W$ is limit-closed, suffix-closed and fusion-closed. Hence, $W$ is $R$-generable.

The first proof of Theorem 5.2.12 describes a way to check if the language accepted by a safety automaton is $R$-generable. This check can be simplified further when the safety automaton is known to be deterministic and irredundant. The following lemma shows that determinism and initial freedom are sufficient conditions to show that a safety automaton is 1-definite.

**Lemma 5.2.13.** If a deterministic safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle$ is initially-free then $A$ is also 1-definite.

**Proof.** A contradiction results from the assumption that $A$ is not 1-definite. Suppose there exists a letter $\sigma \in \Sigma$ and there exist two states $q_1$ and $q_2$ in $A$ such that after reading $\sigma$ the automaton $A$ can be either in state
Figure 5.7: Irredundant automata for three LTL formulae. All states have priority 1. The one on the left is 1-definite because even when the current state of the automaton is not known, reading the input a single letter is enough to determine the next state of the automaton. This automaton shows that both LTL formulae represent transition constraints (transition constraints are defined in Section 5.3). The automaton on the right is not definite because if the current state is not known then reading the input letter \((r \land \neg g)\) is not enough to compute the current state of this automaton.

In summary, an initially-free, deterministic safety automaton accepts an \(\mathcal{R}\)-generable language. However, a redundant deterministic safety automaton may still accept an \(\mathcal{R}\)-generable language but it may not be initially-free or 1-definite. Therefore, if determinism and irredundancy are known to hold then one only needs to check initial freedom to check whether the language accepted by the safety automaton is \(\mathcal{R}\)-generable.

**Example 5.2.14.** Consider the automaton shown on the left in Figure 5.7. This is a reduced automaton that is deterministic, initially-free, and 1-definite. The language accepted by this automaton is \(\mathcal{R}\)-generable and the relation \(R\) is given by the Boolean formula \((\neg r \land \neg r') \lor (\neg r \lor r') \lor (r \land r') \lor (r \land \neg r' \land g')\), which can be simplified to \(\neg r \lor r' \lor g'\).

If a language \(W \subseteq \Sigma^\omega\) is fusion-closed and limit-closed then \(W\) is a subset of some \(\mathcal{R}\)-generable language \(W' \subseteq \Sigma^\omega\). The language of a subset \(R_{in} \subseteq \Sigma\) is \(R_{in} \Sigma^\omega\) denoted by \(L(R_{in})\). The following theorem shows the connec between \(W\), \(W'\), and \(L(R_{in})\).
Theorem 5.2.15. Given a fusion-closed and limit-closed language $W \subseteq \Sigma^\omega$ then there exists an $R$-generable language $W'$ and a set of letters $R_{in} \subseteq \Sigma$ such that $W = W' \cap L(R_{in})$.

Proof. By Corollary 5.2.10 the language $W$ is accepted by a 1-definite safety automaton $A$. One can derive an initially-free, 1-definite safety automaton $A'$ from $A$ through the construction described in the proof of Lemma 5.2.6. (That is, if $A$ is initially free then $A'$ is $A$ otherwise the initial state and outgoing transitions to all the states of $A$ are added–an outgoing transition to the initial state of $A$ is only added if it has a self transition–the clone is then made the initial state of $A'$.) Let $W'$ be the language accepted by $A'$. Since adding transitions cannot remove words from the accepted language, $W \subseteq W'$. By Theorem 5.2.12 $W'$ is an $R$-generable language.

Let $R_{in} = \{ \sigma \mid \exists \sigma w \in W \}$ then $W \subseteq L(R_{in})$. Let $A_{in}$ be the automaton that accepts $L(R_{in})$ then the composition of the two automata $A'$ and $A_{in}$ accept the language $W' \cap L(R_{in})$. Since the composition of $A'$ and $A_{in}$ accept the language of $A$, it follows that $W = W' \cap L(R_{in})$.

The following result is already foreshadowed in [PRS63]. This theorem provides us with a method to detect $\omega$-regular properties which describe $R$-generable languages.

Theorem 5.2.16. If an irredundant safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi \rangle$ that accepts the $\omega$-regular safety property $\varphi$ is not 1-definite, then no other irredundant automaton with the alphabet $\Sigma$ for $\varphi$ is 1-definite.

Proof. A 1-definite automaton must be deterministic. Every state of $A$ is bisimilar to some state of another deterministic automaton $A'$ for $\varphi$. Suppose there is a letter $\sigma \in \Sigma$ such that there are states $q_1$ and $q_2$ in $A$ for which $\delta(q_1, \sigma) = q'_1$, $\delta(q_2, \sigma) = q'_2$, and $q'_1 \neq q'_2$. The states in $A'$ that are bisimilar to $q_1$ and $q_2$ fail the 1-definiteness test as well (because the states of $A'$ bisimilar to $q'_1$ and $q'_2$ cannot be the same state nor can they be bisimilar to each other).

5.3 LTL and $R$-Generability

This section provides a syntactic characterization of LTL formulae that describe $R$-generable languages. An LTL formula that does not satisfy the syntactic check but describes an $R$-generable language
can be converted to an equivalent LTL formula that satisfies the syntactic check through the automata based
approach to detect $\mathcal{R}$-generable languages described in the previous section.

Definition 5.3.1. An LTL formula $\varphi$ is a transition constraint if it belongs to the class defined by the
following grammar:

$$
P ::= G(f),
$$

$$
f ::= p \mid n \mid f \land f \mid f \lor f,
$$

$$
p ::= x \mid \neg x,
$$

$$
n ::= Xp,
$$

where $x$ is a proposition.

The grammar defines LTL formulae in negation normal form. The only permissible temporal operator
inside the $G$ operator is the $X$ operator and the nesting of $X$ is not allowed. The set of safety properties
described by Definition 5.3.1 is closed under conjunction. As one shall see in the remainder of this section,
every $\mathcal{R}$-generable language has an equivalent transition constraint.

Given a formula $\varphi$ produced by the grammar in Definition 5.3.1, the relation $R$ that generates the
language of $\varphi$ is obtained by replacing each subformula $X x$ by $x'$ and each subformula $X \neg x$ by $\neg x'$. Finally,
the $G$ operator is discarded to obtain the propositional formula that is the representation of $R$. Conversely,
given a relation $R \subseteq \Sigma \times \Sigma'$, an LTL safety formula $\varphi_R$ in the form described in Definition (5.3.1) can
be obtained by replacing each $x'$ by $X x$ and finally applying the $G$ operator. (The relation $R \subseteq \Sigma \times \Sigma'$
generated from a transition constraint is a Boolean formula; a minterm that satisfies this formula describes
a pair $(\sigma, \sigma') \in R$ such that the cube of non-primed variables extracted from the minterm encodes $\sigma$ and the
cube of primed variables extracted from the minterm encodes $\sigma'$.)

Example 5.3.2. Example 5.2.14 is continued here. There are LTL formulae that describe transition con-
straints even though they do not satisfy the grammar described in Definition 5.3.1. Simple rewriting
suffices for something like $\varphi = G(r \rightarrow X(r \lor g))$, while the construction of an irredundant safety autom-
on is used to show that $\psi = G(r \rightarrow X(r \mathcal{W} g))$ is equivalent to the transition constraint $\varphi$. (The formula
$r\, W\, g$ is not equivalent to any transition constraint. The automaton for this property is shown on the right in Figure 5.7; one can observe that this automaton is not 1-definite.) Thanks to Theorem 5.2.16, checking the conditions of Theorem 5.2.12 on this irredundant automaton is enough to decide whether the formula is equivalent to a transition constraint or not.

Given an LTL formula $\varphi$ that describes an $R$-generable language but fails the syntactic check, one can always check if $A_\varphi$ is an initially-free, 1-definite safety automaton. Then a relation $R$ can be generated from $A_\varphi$ and $R$ can then be converted to a transition constraint that describes the $R$-generable language specified by the property $\varphi$. In the Section 5.5, it is shown that the realizability of a transition constraint can be checked through the relation $R$. Once the transition constraint is proven to be realizable then an implementation from $R$ can be generated by solving a sequence of Boolean equations.

### 5.4 General Safety Properties

Safety properties like $r\, W\, g$ are neither suffix-closed nor fusion-closed. The objective of this section is to find an $\hat{R}$-generable language $\hat{W}$ that embeds an arbitrary safety language $W$. It is shown that a fusion-closed and limit-closed language $\hat{W}$ over an augmented alphabet $\hat{\Sigma}$ exists such that the language $W$ over the alphabet $\Sigma$ is in one-to-one correspondence with $\hat{W}$. Theorem 5.2.15 can be invoked to decompose $\hat{W}$ into the intersection of an $\hat{R}$-generable language and a language in which only the initial letters are constrained.

Given safety language $W$ that is not $R$-generable, the problem of augmenting the alphabet $\Sigma$ to $\hat{\Sigma} = \Sigma \times K$ is solved through the irredundant automaton $A$ that accepts $W$. Let $A_\varphi = (\hat{\Sigma}, Q, q_{\text{in}}, \delta, \pi)$ be an irredundant deterministic safety automaton that accepts the language described by property $\varphi$. If $A_\varphi$ is not 1-definite then $L(A_\varphi)$ is not fusion-closed, which implies that there exists $\sigma \in \Sigma$ such that the automaton $A_\varphi$ can be in two or more different states after reading the letter $\sigma$. To obtain an irredundant automaton which is 1-definite but retains the transition structure isomorphic to $A_\varphi$, one needs to disambiguate the labels of the transitions. The first step in this process is to identify the ambiguity of the irredundant automaton $A_\varphi$ after reading one letter. This information is then used to create an incompatibility graph. The vertices of this graph are the states of the automaton, and there is an edge between two distinct vertices $v_1$ and $v_2$.
Table 5.1: The description various symbols used in this chapter.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ϕ</td>
<td>ω-regular safety property</td>
</tr>
<tr>
<td>A_ϕ</td>
<td>Irredundant safety automaton for ϕ</td>
</tr>
<tr>
<td>ˆA_ϕ</td>
<td>1-definite automaton derived from A_ϕ by augmenting the alphabet of ϕ</td>
</tr>
<tr>
<td>R_{in}</td>
<td>predicate describing the labels on the outgoing edges of ˆA_ϕ</td>
</tr>
<tr>
<td>R_ϕ</td>
<td>relation describing the valid transitions of ˆA_ϕ</td>
</tr>
<tr>
<td>G_ϕ</td>
<td>Input-based game derived from A_ϕ</td>
</tr>
<tr>
<td>ˆG_ϕ</td>
<td>The game G_ϕ restricted to protagonist’s winning states and strategies</td>
</tr>
<tr>
<td>ˆG_ϕ^R</td>
<td>Input-based game derived from A_ϕ</td>
</tr>
<tr>
<td>ˆG_ϕ^R</td>
<td>The game ˆG_ϕ restricted to protagonist’s winning states and strategies</td>
</tr>
<tr>
<td>G_ϕ^R</td>
<td>Relational game derived from R_{in} and R_ϕ</td>
</tr>
<tr>
<td>ˆG_ϕ^R</td>
<td>The game ˆG_ϕ^R restricted to protagonist’s winning strategies</td>
</tr>
</tbody>
</table>

iff there is a letter σ ∈ Σ such that, for some states t_1 and t_2, δ(t_1, σ) = v_1 and δ(t_2, σ) = v_2. The chromatic number of this incompatibility graph gives the minimum cardinality of the K required to turn the irredundant automaton A_ϕ to a 1-definite irredundant automaton ˆA_ϕ. Each element of K corresponds to one of the colors of a minimum cardinality coloring γ of the incompatibility graph; γ : Q → K maps states to colors. One can obtain from A_ϕ another safety automaton ˆA_ϕ = (ˆΣ, Q, q_in, ˆδ, π), where ˆΣ = Σ × K. The transition function is given by:

$$\hat{δ}(q, (σ, k)) = \begin{cases} δ(q, σ) & k = γ(δ(q, σ)) \text{ and } δ(q, σ) ≠ \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

The label of each transition is augmented with the color of the target state; this guarantees that ˆA_ϕ is a 1-definite safety automaton. (If A_ϕ can be in several different states after reading a letter σ ∈ Σ then all the states are colored differently in the incompatibility graph.) Since ˆA_ϕ is a deterministic 1-definite automaton, the transition function ˆδ can be replaced by a new transition function ˜δ : ˆΣ → Q where ˜δ((σ, k)) = {q′ | ∃q ∈ Q : q′ = δ(q, σ) ∧ k = γ(q′)} (this set is either a singleton or it is empty). The state coloring function γ can also be replaced by an edge coloring function ˜γ : Q × ˆΣ → K, where ˜γ(q, σ) = γ(δ(q, σ)). (The color of an initial state that does not have any incoming transitions is not important.) It can be seen in Section 5.5 that the map ˜γ is convenient when checking realizability of the property ϕ through its transition constraint.
Figure 5.8: (a) The incompatibility graph derived from the irredundant automaton for the LTL formula $r \mathcal{W} g$ shown in Figure 5.7. (b) The incompatibility graph derived from the irredundant automaton for the LTL formula $G(r \land \neg g \to X(r \lor g \lor X(r \lor g)))$ shown in Figure 5.10.

The following example describes the incompatibility graphs obtained from two irredundant automata that accept safety languages that are not fusion-closed.

**Example 5.4.1.** Consider the irredundant automaton for $r \mathcal{W} g$ shown on the left in Figure 5.9. The 1-definite automaton over the augmented language is shown on the left in Figure 5.10. The input $r \land \neg g$ is the reason why the two states are incompatible. The chromatic number of this graph is 2. The incompatibility graph is shown on the left in Figure 5.8.

Consider the irredundant automaton for $G(r \land \neg g \to X(r \lor g \lor X(r \lor g)))$ shown on the right in Figure 5.9. The 1-definite automaton over the augmented language is shown on the right in Figure 5.10. The input $\neg r \land \neg g$ is the reason why $q_{in}$ and $q_2$ are incompatible. The chromatic number of this graph is also 2. The incompatibility graph is shown on the right in Figure 5.8.

Figure 5.9: Automata over $\Sigma$ for $\varphi_1 = r \mathcal{W} g$ and $\varphi_2 = G(r \land \neg g \to X(r \lor g \lor X(r \lor g)))$
The automaton $\hat{A}_\varphi$ is derived from an irredundant automaton $A_\varphi$ through the coloring procedure described earlier. The automaton accepts a fusion-closed and limit-closed language over the alphabet $\hat{\Sigma}$. The language of $\hat{A}_\varphi$ can be represented by a relation $R_\varphi$ and an initial predicate $R_{\text{in}}$. Let $\zeta: \hat{\Sigma} \to \Sigma$ be the projection function that maps letter $(\sigma, k) \in \hat{\Sigma}$ to $\sigma$; let $\zeta(w)$ and $\zeta(W)$ denote the point-wise extensions of $\zeta$ to a word $w \in \hat{\Sigma}^\omega$ and to a language $W \subseteq \hat{\Sigma}^\omega$. The following lemma shows that the safety language accepted by $A_\varphi$ is embedded in the language accepted by $\hat{A}_\varphi$. It proves that every word in $L(A_\varphi)$ has a corresponding word in $L(\hat{A}_\varphi)$ through the runs of the automata $A_\varphi$ and $\hat{A}_\varphi$. The projection of language of $\hat{A}_\varphi$ on the alphabet $\Sigma$ is such that $\zeta(L(\hat{A}_\varphi)) = L(\varphi)$.

**Lemma 5.4.2.** Given a safety property $W \subseteq \Sigma^\omega$, there exists an augmented alphabet $\hat{\Sigma}$ and an $R$-generable language $\hat{W} \subseteq \hat{\Sigma}^\omega$ such that $\zeta: \hat{\Sigma}^\omega \to \Sigma$ is a bijection from $\hat{W}$ to $W$.

**Proof.** Let $A_\varphi$ be an irredundant deterministic safety automaton accepting $W$; let $\hat{A}_\varphi$ be the 1-definite automaton obtained through the procedure described above. Let $\hat{W}$ be the language of $\hat{A}_\varphi$. The automata $A_\varphi$ and $\hat{A}_\varphi$ are isomorphic and every edge $(q, \sigma)$ of $A_\varphi$ has a unique corresponding edge $(q, (\sigma, \hat{\gamma}(q, \sigma)))$ in $\hat{A}_\varphi$. Therefore, for every word $w \in W$, there is a unique word $\hat{w} \in \hat{\Sigma}^\omega$ such that $\zeta(\hat{w}) = w$ and $\hat{w}$ has a run in $\hat{A}_\varphi$. This run $\hat{\rho}$ is identical to the run $\rho$ of $A_\varphi$ when it reads $w$, which implies $\hat{w} \in \hat{W}$. Since for every word $w \in W$, $|\zeta^{-1}(w) \cap \hat{W}| = 1$ (it cannot be greater than 1 as that would imply a word is mapped to two unique words, and it cannot be 0 as that would imply existence of a word that does not have a corresponding word in $\hat{W}$), the restriction of the function $\zeta$ to $\hat{W} \subseteq \hat{\Sigma}$ is a bijection from $\hat{W}$ to $W$. 

**Example 5.4.3.** Consider the property $\varphi_1 = r \lor g$. The alphabet $\Sigma$ is $2^{\{r,g\}}$ and the only letter that may lead to more than one state is $r \land \neg g$. The irredundant automaton for $\varphi_1$ is shown in Figure 5.7. The two states of the automaton $A_\varphi$ are incompatible and the chromatic number of the incompatibility graph is 2 (as described in Example 5.4.1). Therefore, $|K| = 2$ and one can choose $K = \{1, 2\}$ where 1 is encoded by $x$ and 2 is encoded by $\neg x$. The resulting automaton is shown on the left in Figure 5.10. According to Theorem 5.2.12 the language of $r \lor g$ is the projection on $2^{\{g,r\}}$ of the language of $\hat{A}_{\varphi_1}$, where
L(A_\phi_1) = L(R_{in}) \cap L(R_\phi_1) and

R_{in} = ((r \land \neg g \land \neg x) \lor (g \land x))

R_\phi_1 = (r \land \neg g \land \neg x \land r' \land \neg g' \land \neg x') \lor (r \land \neg g \land \neg x \land g' \land x') \lor (x \land x') .

**Example 5.4.4.** Consider the LTL formula \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \), which is not a transition constraint. The alphabet \( \Sigma_\varphi = 2^{r,g} \) is augmented with the alphabet \( K = \{1, 2\} \), where 1 is encoded with \( y \) and 2 is encoded with \( \neg y \). The automaton \( \hat{A}_{\varphi_2} \) labeled with the augmented alphabet is shown on the right.

**Figure 5.10:** Automata over \( \hat{\Sigma} \) for \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \).

**Figure 5.11:** \( \varphi = \varphi_1 \land \varphi_2 \) where \( \varphi_1 = r W g \) and \( \varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g))) \).
The language of $\phi_2$ is the projection on $2\{g,r\}$ of the language of $R_{in} \land R_{\phi_2}$, with

$$R_{in} = \neg y$$

$$R_{\phi_2} = \left((\neg r \lor g) \land \neg y \land \neg y'\right) \lor$$

$$(r \land \neg g \land \neg y \land (r' \lor g') \land \neg y') \lor$$

$$(r \land \neg g \land \neg y \land \neg r' \land \neg g' \land y') \lor$$

$$(\neg r \land \neg g \land y \land (r' \lor g') \land \neg y') \right).$$

There exists another approach that can derive a transition constraint from an arbitrary safety property [BF97]. This approach converts an LTL safety property to a separated normal form. For instance, $G(r \rightarrow X(r \lor g))$ can be written as $G((r \rightarrow Xx_1) \land (x_1 \rightarrow g \lor (r \land Xx_1)))$, where $x_1$ is an auxiliary variable. This rewriting, however, may use more auxiliary variables than the approach based on the incompatibility graph (as shown in Figure 5.7 this property has an equivalent transition constraint which does not use any auxiliary variables).

Given a conjunction of safety properties, obtaining transition constraints from each of them and then conjoining all the transition constraints does not guarantee optimality (more auxiliary variables may be needed through this approach). This is because the product automaton obtained from composing the irredudant automata for the corresponding safety properties may be neither reduced nor irredundant. In fact, the conjunction of two languages that are not fusion-closed and limit-closed may result in a fusion-closed and limit-closed (and maybe even be suffix-closed) languages. The following example shows that when two languages $W_1$ and $W_2$ are embedded in languages $\hat{W}_1$ and $\hat{W}_2$ respectively then $W_1 \cap W_2$ is embedded in $\hat{W}_1 \cap \hat{W}_2$. However, there may exist a language $\hat{W}$ such that $W_1 \cap W_2$ is embedded in $\hat{W}$ and the alphabet of $\hat{W}$ is smaller than the alphabet of $\hat{W}_1 \cap \hat{W}_2$.

**Example 5.4.5.** Consider the property $\varphi = \varphi_1 \land \varphi_2$, where $\varphi_1 = r \lor g$ and $\varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g)))$. The irredundant automata for LTL formulae $\varphi_1$ and $\varphi_2$ are shown in Figure 5.10. The irredundant automaton for $\varphi_1 \land \varphi_2$ is shown on the left in Figure 5.11, while the automaton obtained from composing the automata for the two properties $\varphi_1$ and $\varphi_2$ is shown on the right in Figure 5.11. This product
automaton is not irredundant because \( q_{in} \) and \( q_1 \) are bisimilar states. The irredundant automaton \( A_\varphi \) shown on the left in Figure 5.11 can be obtained from the product automaton.

The relation \( R_\varphi = R_{\varphi_1} \land R_{\varphi_2} \), where \( R_{\varphi_1} \) is given in Example 5.4.3 and \( R_{\varphi_2} \) is given in Example 5.4.4; four current state variables \( \{r, g, x, y\} \) appear in the support of the relation \( R_\varphi \). However, the incompatibility graph of the irredundant automata \( A_\varphi \) has chromatic number 2 (\( q_{in} \) conflicts with \( q_2 \) because of the input \( r \land \neg g \) and \( q_1 \) conflicts with \( q_3 \) because of the input \( \neg r \land \neg g \). The incompatibility graph has four nodes and two edges. The graph has two components with two nodes each. Therefore it can be colored with two colors only.). Hence, the alphabet \( \Sigma_\varphi \) only needs to be augmented with one binary \( y \) variable, which encodes the 2 colors \( \{1, 2\} \) (the encoding has been discussed in examples 5.4.3 and 5.4.4). The language \( \hat{W} \) which embeds \( L(\varphi) \) is given by the relation \( R'_\varphi \) and an initial predicate \( R'_{in} \), where

\[
R'_{in} = (r \lor g) \land \neg y
\]

\[
R'_\varphi = (r \land \neg g) \land \neg y \land (r' \lor g') \land \neg y' \lor
\]

\[
(r \land \neg g \land y \land \neg g \land y) \lor
\]

\[
(r \land \neg g \land y \land g \land \neg y) \lor
\]

\[
(\neg r \land \neg g \land y \land r \land \neg g \land y) \lor
\]

\[
(\neg r \land \neg g \land y \land g \land \neg y) \land
\]

There are only 3 current state variables in the support of the relation \( R'_\varphi \) generated from \( A_\varphi \). In this particular case, the automaton on the right in Figure 5.11, which is not irredundant also yields the same language \( \hat{W} \). However, this is not always true and irredundant automaton is often necessary to find the smallest alphabet for the purpose of augmentation.

The above example demonstrates that when augmenting the alphabet of conjunction of two safety properties which describe languages that are not \( \mathcal{R} \)-generable, the best results can only be achieved if the automaton for the conjunction is irredundant. One can either conjoin the property and then obtain an irre-
dundant automaton or conjoin the irredundant automata for each property and then obtain an irredundant automaton from the resulting automaton (the composition of two irredundant automaton is not guaranteed to be irredundant.) However, this may not be practically feasible and therefore the current approach generates irredundant automaton for each property and then obtains a transition constraint from it.

In this section, it has been shown that any \( \omega \)-regular safety property \( \varphi \) can be embedded in another fusion-closed and limit-closed language. Whenever a specification contains a large number of safety properties, this method is able to generate a significantly more compact transition constraint. This transition constraint requires fewer symbolic variables and on top of that its encoding does not contain a lot of redundancies. The next section describes how to check the realizability of any safety property through its transition constraint.

### 5.5 Realizability of Relation-Based game \( \hat{G}^R_{\varphi} \)

This section describes a procedure that checks the realizability of a safety property \( \varphi \) embedded in a fusion-closed, limit-closed language \( \hat{W} \). The language \( \hat{W} \) is described by an initial predicate \( R_{in} \) and relation \( R_{\varphi} \). One can obtain an input-based game \( G_{\varphi} \) from the automaton \( A_{\varphi} \) that recognizes the language described by \( \varphi \). One can also derive and input-based game \( \hat{G}_{\varphi} \) that is derived from the automaton that accepts \( \hat{W} \). Finally a game \( \hat{G}^R_{\varphi} \), can be derived from \( R_{\varphi} \) and \( R_{in} \). It is first shown that one can obtain system’s or environment’s winning strategies for \( G_{\varphi} \) by playing \( \hat{G}_{\varphi} \) and vice-versa. Then it is shown that one can obtain system’s or environment’s winning strategies for \( \hat{G}_{\varphi} \) by playing \( \hat{G}^R_{\varphi} \) and vice-versa. Therefore, one can obtain system’s or environment’s winning strategies for \( G_{\varphi} \) by playing \( \hat{G}^R_{\varphi} \).

The game-theoretic approach to checking realizability of an LTL property is now briefly reviewed. Let \( \varphi \) be an LTL safety formula defined over the set of Boolean variables \( X = X_{ed} \cup X_s \cup X_{ep} \). One can obtain the automata \( A_{\varphi} \) and \( \hat{A}_{\varphi} \) as described in the previous section, where \( A_{\varphi} \) accepts the language \( L(\varphi) \), which is embedded in the language accepted by \( \hat{A}_{\varphi} \). (As mentioned earlier, \( \Sigma = \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \), while \( \hat{\Sigma} = \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \times \Sigma_K \), where \( \Sigma_{ed} = 2^{X_{ed}}, \Sigma_s = 2^{X_s}, \Sigma_{ep} = 2^{X_{ep}}, \) and \( \Sigma_K = 2^{X_K} \).\(^5\) These two automata

---

\(^5\) The coloring function \( \gamma(Q) \) is written for the point-wise extension of \( \gamma \) to set of states \( Q \subseteq Q \), then it may be the case that \( \gamma(Q) \subseteq \Sigma_K \). As it shall be seen later in the chapter that this does not create an issue when solving these games.
can be interpreted as input-based games $G_\varphi = (\Sigma, Q, q_{in}, \delta, \pi)$ and $\hat{G}_\varphi = (\hat{\Sigma}, Q, q_{in}, \hat{\delta}, \pi)$ respectively. In $G_\varphi$ the system controls the signals in $X_s$ and the environment controls the signals in $X_{ed} \cup X_{ep}$. In $\hat{G}_\varphi$ the system controls the signals in $X_s \cup X_K$, while the environment controls the signals in $X_{ed} \cup X_{ep}$. The reason behind the partitioning of signals controlled by the environment was discussed in Section 2.7. The choice of the color from $\Sigma_K$ is given to the system because it is the system’s objective to satisfy the property $\varphi$. This choice is what makes the two games $G_\varphi$ and $\hat{G}_\varphi$ interchangeable. In both games $G_\varphi$ and $\hat{G}_\varphi$, the objective of the protagonist is to satisfy the safety condition by forcing an infinite play of the game, while the objective of the antagonist is to prevent the protagonist from playing endlessly.

When the token is in state $q \in Q$, the environment chooses a letter $\sigma_{ed}$ and discloses it to the system; then the system chooses a letter $\sigma_s$ and discloses it to the environment; then the environment selects a letter $\sigma_{ep}$; finally the value of $\sigma_K$ is evaluated by the system through the function $\hat{\gamma}(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))$ and the token moves to state $q' = \hat{\delta}(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}, \sigma_K))$. In the game $G_\varphi$ the system’s strategy is given by $\tau_s: Q \times \Sigma_{ed} \times S \rightarrow \Sigma_s \times S$. While in the game $\hat{G}_\varphi$ the system’s strategy is $(\hat{\tau}_s, \hat{\tau}_K)$. The first component $\hat{\tau}_s$ is identical to $\tau_s$, while $\hat{\tau}_K : Q \times \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \times S \rightarrow K \times S$ is the additional component, where $\hat{\tau}_K(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}), s) = (\hat{\gamma}(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}), s), \hat{\gamma}(\sigma_K, (\sigma_{ed}, \sigma_s, \sigma_{ep}), s))$, which is the unique value of the letter $\sigma_K$—the color of the target state.\(^6\) It may seem that the system has extra freedom by computing $\hat{\gamma}$, however this is not true because for every state $q$ and letter $\sigma$ the value of $\hat{\gamma}(q, \sigma)$ is unique. Therefore, it is convenient to assign the task of computing this function to the system. (The environment cannot be entrusted with this task because it will allow the environment to win the game by choosing incorrect values for $\hat{\gamma}(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}), s)$.) In summary, given $(\tau_s, \hat{\gamma})$ one can obtain the pair of strategies $(\hat{\tau}_s, \hat{\tau}_K)$ and vice versa.

The following lemma proves that a winning strategy of the system in $\hat{G}_\varphi$ is a winning strategy in $G_\varphi$ and a winning strategy of the system in $G_\varphi$ can be used to generate a winning strategy for the system in $\hat{G}_\varphi$ in a straightforward manner.

**Lemma 5.5.1.** $\tau_s$ is a winning strategy for the system in $G_\varphi$ iff the corresponding pair of strategies $\hat{\tau}_s$ and $\hat{\tau}_K$ is a winning strategy for the system in $\hat{G}_\varphi$.

---

\(^6\) In this game the system has the option to record the environment’s choice of letter from $\Sigma_{ep}$, however having this knowledge does not give the system an advantage when making its selection from $\Sigma_s$ because the choice of $\Sigma_{ep}$ in the previous step has no effect on the priority of the previous state. Therefore, it is assumed that the system does not update its memory after choosing the correct color from $\Sigma_K$. 
\( \hat{\tau}_K \) are winning in \( \hat{G}_\varphi \).

**Proof.** Suppose \( \tau_s \) is a winning strategy for the system in \( G_\varphi \). Suppose the environment’s choices at step \( i \) of the game \( \hat{G}_\varphi \) are given by the pair of functions \( \epsilon_{ed} : \mathbb{N} \to \Sigma_{ed} \) and \( \epsilon_{ep} : \mathbb{N} \to \Sigma_{ep} \). If the environment plays these choices in \( G_\varphi \) then the word \( w \) is generated. This word \( w \) produces a run \( \rho \) in game \( G_\varphi \). Since \( \tau_s \) is a winning strategy the run \( \rho \) is a winning run for the system in \( G_\varphi \). Suppose in game \( \hat{G}_\varphi \) the system plays with the strategies \( \hat{\tau}_s \) and \( \hat{\tau}_K \) obtained from \( \tau_s \) against the environment’s choices given by \( \epsilon_{ed} \) and \( \epsilon_{ep} \); then by Lemma 5.4.2 the word \( \hat{w} \) is produced such that \( \hat{w} = \zeta^{-1}(w) \cap \hat{W} \). The run of \( \hat{G}_\varphi \) when reading \( \hat{w} \) is identical to \( \rho \) and such a run is winning in \( \hat{G}_\varphi \) as well. Hence, \( \tau_s \) is a winning strategy in \( G_\varphi \). The proof in the other direction follows the same reasoning.

It has been shown that one can compute the system’s strategies in \( G_\varphi \) by playing the game \( \hat{G}_\varphi \). Now it will be shown that one can compute the system’s winning strategies for \( \hat{G}_\varphi \) by playing a game obtained from \( R_\varphi \) and \( R_{in} \), where \( R_\varphi \) is a relation and \( R_{in} \) is an initial predicate. From Theorem 5.2.15 and Lemma 5.4.2 one observes that \( L(\hat{A}_\varphi) = L(R_\varphi) \cap L(R_{in}) \).

It is now shown that the realizability of \( \varphi \) can be checked implicitly through predicate \( R_{in} \) and relation \( R_\varphi \) without the construction of \( \hat{G}_\varphi \). This is done by establishing the correspondence between \( \hat{G}_\varphi \) and the game between system and environment \( \hat{G}_\varphi \) defined as follows. In every round, the letter \( \hat{w}_i = (\sigma_{ed}^i, \sigma_s^i, \sigma_{ep}^i, \sigma_K^i) \) is selected when the environment chooses a letter \( \sigma_{ed}^i \) and discloses it to the system; then the system chooses a letter \( \sigma_s^i \) and discloses it to the environment; then the environment selects a letter \( \sigma_{ep}^i \); finally the system computes \( \sigma_K^i \) through the function \( \hat{\gamma} \). For \( i > 0 \), the value of \( \sigma_K^i \) is given by the function \( \hat{\gamma}(q_{i-1}, \sigma) \), which can be computed from \( \hat{w}_{i-1} \) and \( (\sigma_{ed}^i, \sigma_s^i, \sigma_{ep}^i) \) as \( \sigma_K^i = \hat{\gamma}(\hat{\delta}(\hat{w}_{i-1}), (\sigma_{ed}^i, \sigma_s^i, \sigma_{ep}^i)) \); \( \sigma_0^K \) is computed by \( \hat{\gamma}(q_{fin}, \sigma^0) \). The system wins if \( \hat{w}_0 \) satisfies \( R_{in} \) and for \( i > 0 \), \( (\hat{w}_{i-1}, \hat{w}_i) \in R_\varphi \). The system’s strategy in the game \( \hat{G}_\varphi \) is a pair of functions

\[
\hat{\tau}_s^R : S \times \hat{\Sigma} \times \Sigma_{ed} \rightarrow \Sigma_s \times S
\]

\[
\hat{\tau}_K^R : S \times \hat{\Sigma} \times \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \rightarrow \Sigma_K \times S ,
\]

where the system’s strategy \( \tau_K^R(s, \hat{\sigma}, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \) is of the form \( (\hat{\gamma}(\hat{\delta}(\hat{\sigma}), (\sigma_{ed}, \sigma_s, \sigma_{ep})), s') \). The first letter is computed with the memory initialized to \( s_{in} \). The following discussion establishes the correspondence
between the games $\hat{G}_\varphi$ and $\hat{G}^R_\varphi$ by proving that a system’s strategy in one of the games is winning iff it is winning in the other game.

**Definition 5.5.2.** Let the pair of functions $(\hat{\tau}_s, \hat{\tau}_K)$ be a system’s strategy in $\hat{G}_\varphi$ and let the augmented memory be $S = S \times Q$; then the augmented strategy is $(T_s, T_K) = \text{AUG}((\hat{\tau}_s, \hat{\tau}_K))$ where $T_s : S \times \Sigma_{ed} \to \Sigma_s \times S$ and $T_K : S \times \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \to \Sigma_K \times S$ given by

\[
T_s((s, q), \sigma_{ed}) = (\sigma_s, (t, q)) \quad \text{where} \quad (\sigma_s, t) = \hat{\tau}_s(s, q, \sigma_{ed})
\]

\[
T_K((s, q), \sigma_{ed}, \sigma_s, \sigma_{ep}) = (\sigma_K, (t, v)) \quad \text{where} \quad (\sigma_K, t) = \hat{\tau}_K(s, q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \quad \text{and} \quad v = \delta((\sigma_{ed}, \sigma_s, \sigma_{ep}, \sigma_K)) .
\]

Let the pair of functions $(\hat{\tau}_s^R, \hat{\tau}_K^R)$ be a system’s strategy in $\hat{G}^R_\varphi$ and let the augmented memory be $S = S \times \hat{\Sigma}$; then the augmented strategy $(T_s, T_K) = \text{AUG}((\hat{\tau}_s^R, \hat{\tau}_K^R))$ is given by

\[
T_s((s, \sigma), \sigma_{ed}) = (\sigma_s, (t, \sigma)) \quad \text{where} \quad (\sigma_s, t) = \hat{\tau}_s^R(s, \sigma, \sigma_{ed})
\]

\[
T_K((s, \sigma), \sigma_{ed}, \sigma_s, \sigma_{ep}) = (\sigma_K, (t, \sigma')) \quad \text{where} \quad (\sigma_K, t) = \hat{\tau}_K^R(s, \sigma, \sigma_{ed}, \sigma_s, \sigma_{ep}) \quad \text{and} \quad \sigma' = (\sigma_{ed}, \sigma_s, \sigma_{ep}, \sigma_K) .
\]

The following lemmas prove that one can compute system’s winning strategies for $\hat{G}_\varphi$ from $\hat{G}^R_\varphi$ and vice versa.

**Lemma 5.5.3.** $(T_s, T_K)$ is a system’s winning strategy in the game $\hat{G}_\varphi$ iff it is also winning in the game $\hat{G}^R_\varphi$.

**Proof.** Let the pair of functions $\epsilon_{ed} : \mathbb{N} \to \Sigma_{ed}$ and $\epsilon_{ep} : \mathbb{N} \to \Sigma_{ep}$ be the environment’s choice of letters for the $i$th turn in $\hat{G}_\varphi$. Then the word generated by $(\epsilon_{ed}, T_s, \epsilon_{ep}, T_K)$ is $\hat{w}$. If the functions $(\epsilon_{ed}, T_s, \epsilon_{ep}, T_K)$ are used to play the game $\hat{G}^R_\varphi$, then the word generated is also $\hat{w}$. From the construction of $\hat{G}_\varphi$ and $\hat{G}^R_\varphi$ one concludes that $\hat{w} \in L(\hat{G}_\varphi)$ iff $\hat{w} \in L(\hat{G}^R_\varphi)$. (The construction of $\hat{A}_\varphi$, the relation $R_\varphi$, and the predicate $R_{in}$ has been discussed in the previous sections. The interpretation of $\hat{A}_\varphi$ as a game is denoted as $\hat{G}_\varphi$ and
(R_ϕ, R_{in}) is interpreted as a game \( \hat{G}^R_ϕ \). Hence, \((T_s, T_K)\) is a system’s winning strategy in the game \( \hat{G}^{G}_ϕ \) iff it is also winning in the game \( \hat{G}^R_ϕ \). The proof in the other direction follows a similar argument.

Lemma 5.5.4. Given the system’s strategy \((\hat{\tau}_s, \hat{\tau}_K)\) in the game \( \hat{G}_ϕ \) then \((\hat{\tau}_s, \hat{\tau}_K)\) is winning iff the augmented strategy \((T_s, T_K) = \text{AUG}((\hat{\tau}_s, \hat{\tau}_K))\) is also winning in \( \hat{G}_ϕ \).

Proof. Let the pair of functions \( \epsilon_{ed} : \mathbb{N} \rightarrow \Sigma_{ed} \) and \( \epsilon_{ep} : \mathbb{N} \rightarrow \Sigma_{ep} \) be the environment’s choice of letters for the \( i \)th turn. Given the functions \((\epsilon_{ed}, \hat{\tau}_s, \epsilon_{ep}, \hat{\tau}_K)\), let \( \hat{w} \) be the word generated by in \( \hat{G}_ϕ \). Then by Definition 5.5.2 \((\epsilon_{ed}, T_s, \epsilon_{ep}, T_K)\) produce the same word \( \hat{w} \) in \( \hat{G}_ϕ \). Hence, the strategy \((\hat{\tau}_s, \hat{\tau}_K)\) is winning iff \((T_s, T_K)\) is also winning in the game \( \hat{G}_ϕ \).

Lemma 5.5.5. Given system’s strategy \((\hat{\tau}^R_s, \hat{\tau}^R_K)\) in the game \( \hat{G}^R_ϕ \) then \((\hat{\tau}^R_s, \hat{\tau}^R_K)\) is winning iff augmented strategy \((T_s, T_K) = \text{AUG}((\hat{\tau}^R_s, \hat{\tau}^R_K))\) is also winning in \( \hat{G}^R_ϕ \).

Proof. Let the pair of functions \( \epsilon_{ed} : \mathbb{N} \rightarrow \Sigma_{ed} \) and \( \epsilon_{ep} : \mathbb{N} \rightarrow \Sigma_{ep} \) be the environment’s choice of letters in \( i \)th turn. Given the functions \((\epsilon_{ed}, \hat{\tau}^R_s, \epsilon_{ep}, \hat{\tau}^R_K)\) let \( \hat{w} \) be the word generated by in \( \hat{G}^R_ϕ \). Then by Definition 5.5.2 \((\epsilon_{ed}, T_s, \epsilon_{ep}, T_K)\) produce the same word \( \hat{w} \) in \( \hat{G}^R_ϕ \). Hence, the strategy \((\hat{\tau}^R_s, \hat{\tau}^R_K)\) is winning iff \((T_s, T_K)\) is also winning in the game \( \hat{G}^R_ϕ \).

In this section, the correspondence between the games \( G_ϕ, \hat{G}_ϕ, \) and \( \hat{G}^R_ϕ \) has been established. Furthermore, it has been shown that one can compute the system’s winning strategy for \( G_ϕ \) and \( \hat{G}_ϕ \) from the relation-based game \( \hat{G}^R_ϕ \).

5.6 Solving the Relation-Based game \( \hat{G}^R_ϕ \)

This section discusses the operators \( M_{\varphi} \) and \( M_{G_\varphi} \) that are needed to solve the relation-based game \( \hat{G}^R_ϕ \). The main advantage of playing a relation-based game is that it scales better than automata-based games as the number of safety properties increases. Secondly, a compact representation of a parameterized transition function that guarantees the satisfaction of the safety property \( \varphi \) can be obtained through this approach. When the specification contains progress properties then this parameterized transition function is necessary for finding winning strategies for those properties that are also winning for the safety part of the
specification. Parameterization is useful because one should not choose a strategy prematurely, lest the only strategies good for the progress properties should be thrown away.

It is first shown how to check for the existence of winning strategies in $\hat{G}_\varphi^R$ symbolically; that is, by an algorithm that manipulates the characteristic functions of sets and relations over $\hat{\Sigma}$ (this has been discussed in Section 2.8). The game $\hat{G}_\varphi^R$ is played in two stages; the first stage checks the realizability of $R_\varphi$ and the second stage checks the realizability of $R_{in} \land R_\varphi$. Winning strategies are computed once realizability has been established. Given a set of target letters $T(X')$, that is, a set expressed in terms of next-state variables, the pre-image operator $MX$ is defined as follows:

$$MX_\varphi T = \forall X'_{ed} \cdot \exists X'_{s} \cdot \forall X'_{ep} \cdot \exists X'_{K} \cdot R_\varphi(X, X') \land T(X')$$

The set of letters $MX_\varphi T$ is such that for each letter in this set, no matter how the environment chooses its letters, the system can always force the choice of letter from $T(X')$ such that $R_\varphi$ is satisfied. When it is the environment that has to satisfy the property $\varphi$ then the definition of the $MX_\varphi$ operator is as follows:

$$MX_\varphi T = \exists X'_{ed} \cdot \forall X'_{s} \cdot \exists X'_{ep} \cdot \exists X'_{K} \cdot R_\varphi(X, X') \land T(X')$$

Notice that the quantifiers are inverted except the last one and that the order of quantification remains the same. This is because the environment is now the one trying to satisfy the property, while the system and environment make the selections of their respective alphabets in the same order. This second version of the $MX_\varphi$ operator is needed in Section 5.10, where the system has to satisfy its safety property $\varphi$ only when the environment satisfies its own safety property $\psi$.

The greatest fixpoint operator $MG_\varphi$ is defined as $MG_\varphi T = \nu Z . T \land MX_\varphi Z$. The realizability of $R_\varphi$ is checked by computing the realizable subset $\hat{\Sigma}^\varphi$ of $\hat{\Sigma}$ such that $\hat{\Sigma}^\varphi = MG_\varphi T$. The greatest fixpoint computation removes the terminal letters from the alphabet $\hat{\Sigma}$; the terminal letters of the alphabet are defined inductively as letters after which there does not exist a strategy to pick a next letter such that $R_\varphi$ is satisfied or letters after which only terminal letters can be selected to satisfy $R_\varphi$.

Finally, the realizability of $R_{in} \land R_\varphi$ is checked. The system wins the game $\hat{G}_\varphi^R$ if $MX_{in} MG_\varphi T = T$, where

$$MX_{in} T = \forall X'_{'ed} \cdot \exists X'_{s} \cdot \forall X'_{ep} \cdot \exists X'_{K} \cdot R_{in}(X') \land T(X')$$
For every letter in $\hat{\Sigma}_\phi^r$, the system can always pick the next letter from the same set so that $R_\phi$ is satisfied.

The operator $M_{X_{in}}$ establishes the system’s ability to start a word from a letter in $\hat{\Sigma}_\phi^r$ such that $R_{in}$ is satisfied. Therefore, the system wins the game $\hat{G}_{\phi}^R$ iff $M_{X_{in}} M_{G_\phi} \top = \top$, which means that the system can force the selection of a letter from $\hat{\Sigma}_\phi^r$ such that the predicate $R_{in}$ is also satisfied.

Example 5.6.1. This example is the continuation of examples 5.4.3, 5.4.4, and 5.4.5 from Section 5.4.

Consider the property $\varphi = \varphi_1 \land \varphi_2$, where $\varphi_1 = r \bigvee g$ and $\varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g)))$. The system’s input is $r$, while $g$ is the environment’s private input. The property $\varphi$ implies that initially $r$ is asserted until $g$ is asserted; after that, whenever $r$ is asserted then it must be reasserted at least every other step until $g$ is asserted again. $A_{\varphi_1}$ and $A_{\varphi_2}$ are shown in Figure 5.10. With $X_{ed} = \emptyset$, $X_s = \{r\}$, $X_{ep} = \{g\}$, and $X_K = \{x, y\}$, the initial predicate $R_{in}$ and the relation $R_\phi$ are given below:

$$R_{in} = ( (r \land \neg g \land \neg x) \lor (g \land x) ) \land \neg y \quad \text{and} \quad R_\phi = ( (r \land \neg g \land \neg x \land g' \land x') \lor (r \land \neg g \land \neg x \land r' \land \neg g' \land \neg x') \lor (x \land x') ) \land$$

$$((r \land \neg g \land \neg y \land (r' \land g') \land \neg y')) \lor$$

$$((\neg r \land g) \land \neg y \land \neg y') \lor$$

$$(r \land \neg g \land \neg y \land \neg r' \land \neg g' \land y') \lor$$

$$(\neg r \land \neg g \land y \land (r' \land g') \land \neg y')) \ .$$

The iterates of the $M_{G_\phi} \top$ computation are:

$$Z_0 = \top \ , \quad Z_1 = (x \land \neg y) \lor (x \land \neg r \land \neg g) \lor (\neg y \land r \land \neg g) \ , \quad \hat{\Sigma}_\phi^r = Z_2 = Z_1 .$$

Since $M_{X_{in}} Z_2 = \top$, the property $\varphi$ is realizable.

Once the realizability of $\varphi$ is confirmed, one can synthesize a machine whose state graph is similar to the game $\hat{G}_\varphi$ restricted to the system’s winning states. The synthesis procedure for this machine is described in the next section.
5.7 Synthesis from Transition Constraints

This section discusses the final step in the conversion of the sequential synthesis problem of a safety property to a combinational synthesis problem. After the realizability of an \( \omega \)-regular safety property \( \varphi \) is established, one may need to either generate the parameterized representation of a transition function or an implementation that guarantees the satisfaction of this property. The parameterized representation of a transition function that satisfies the given safety properties can then be used to find the strategies that work also for the progress properties of the specification. Obtaining some implementation from this representation is a trivial step; however, getting a small implementation (one that requires little combinational and sequential logic) requires extra effort. One heuristic to overcome this challenge is discussed in Section 5.12.

The parameterized representation of a transition function that satisfies \( \varphi \) is derived from the solution of several Boolean equations. The solutions of the equations derived from \( R_{in} \) guarantee the satisfaction of the initial condition of the implementation. The solutions of the equations derived from \( R_{\varphi} \) guarantee the steady state behavior of the implementation (the satisfaction of \( R_{\varphi} \)). The steady state behaviour is the response of the system once it leaves its initial state.

The parameterized representation of a transition function that guarantees the satisfaction of a safety property \( \varphi \) consists of the initial and steady state solutions for variables in \( X_s' \cup X_K' \) and a state variable \( I \) with the initial value \( \perp \) and then \( \top \) forever. This initialization bit is used to distinguish between the initial and steady state solutions. For each element \( u' \in (X_s' \cup X_K') \) let \( u'_\text{in} \) be the initial value solution and \( u'_\infty \) be the steady state solution. The value of the variables \( u' \in (X_s' \cup X_K') \) is given by \( (\neg I \land u'_\text{in}) \lor (I \land u'_\infty) \).

While the variables \( X \) represent latches (because the variables are the previous value of the corresponding variables in \( X' \)). These variables are initialized to \( \perp \) (because the initial value of variables in \( X'_s \cup X'_K \) do not depend on the initial value of variables in \( X \)) while the update of these variables is provided by the corresponding variables in \( X' \). When the parametric representation does not contain the initialization bit then the latches representing the variables \( X \) are initialized by combinational logic representing \( R_{in} \cap \Sigma_{\varphi}^p \).

It has been shown in Section 5.4 that given any safety property \( \varphi \) defined over the alphabet \( 2^X \) (where \( X = X_{ed} \cup X_s \cup X_{ep} \) is the set of input and output binary variables), one can embed \( L(\varphi) \) in
a fusion-closed and limit-closed language \( \hat{W} \) defined over \( 2^\hat{X} \) where \( X \subseteq \hat{X} \). The language \( \hat{W} \) can be represented as a predicate \( R_{\text{in}}(\hat{X}) \) and a relation \( R_\varphi(\hat{X}, \hat{X}') \). As mentioned in Section 5.2, the variables in \( \hat{X} = X_{\text{ed}} \cup X_s \cup X_{\text{ep}} \cup X_K \) are the previous values of the corresponding signals in \( \hat{X}' = X'_{\text{ed}} \cup X'_s \cup X'_{\text{ep}} \cup X'_{K} \).

When \( \varphi \) is realizable the system has a strategy to force the selection of a new letter \( \hat{\sigma}' \) from the set \( \hat{\Sigma}_r \) such that if the given letter \( \hat{\sigma} \) also belongs to \( \hat{\Sigma}_r \) then \( R_\varphi(\hat{\sigma}, \hat{\sigma}') \) is satisfied. So if the system can force the selection of the initial letter from the set \( \Sigma_r \) such that \( R_{\text{in}} \) is satisfied then the system has a strategy to always force the selection of a letter from \( \Sigma_r \), thus guaranteeing the satisfaction of \( \varphi \). The following two Boolean equations are used to compute the strategy through which the system can select an initial letter from \( \Sigma_r \) such that \( R_{\text{in}} \) is satisfied.

\[
\exists X'_K \cdot I(\hat{X}') = I(\hat{X}') \quad \text{where} \quad I(\hat{X}') = R_{\text{in}}(\hat{X}') \land \hat{\Sigma}_r(\hat{X}') \quad (5.1)
\]
\[
\exists X'_s \cdot I_s(X'_{\text{ed}} \cup X'_s) = I_s(X'_{\text{ed}} \cup X'_s) \quad \text{where} \quad I_s(X'_{\text{ed}} \cup X'_s) = \forall X'_{\text{ep}} \cdot \exists X'_K \cdot I(\hat{X}') \quad (5.2)
\]

In the first equation \( X'_K \) is the set of unknown variables and \( \hat{X}' \setminus X'_K \) is the set of independent variables. In the second equation \( X'_s \) is the set of unknown variables and \( X'_{\text{ed}} \) is the set of independent variables. The solution of the color variables \( X'_K \) depends on the set of variables \( X'_{\text{ed}} \cup X'_s \cup X'_{\text{ep}} \). However, the solution for the system variables \( X'_s \) can only depend on the set of variables \( X'_{\text{ed}} \) as the variables in \( X'_{\text{ep}} \) are private to the environment and the system has to select a letter from \( \Sigma_s \) without knowing the environment’s choice of letter from \( \Sigma_{ep} \). This is the reason for the existential quantification of \( X'_K \) and universal quantification of \( X'_{ep} \) in the second Boolean equation. Once the solution of \( X'_s \) variables is computed, the variables \( X'_s \) are replaced by their solution resulting in the solution of the variables \( X'_K \) to depend only on \( X'_{\text{ed}} \cup X'_{\text{ep}} \). This is because the objective is to find a solution for the \( X'_s \) variables in terms of \( X'_{\text{ed}} \) and \( X'_{ep} \).

To compute the steady state solution one assumes that the current letter \( \hat{\sigma}_i \) is given. Then the following two Boolean equations are used to compute the strategy through which the system can select a new letter \( \hat{\sigma}_{i+1} \) such that the relation \( R_\varphi(\hat{\sigma}_i, \hat{\sigma}_{i+1}) \) is satisfied. The solution to Boolean equations is only valid if the letter \( \hat{\sigma}_i \) is from the set of letters \( \hat{\Sigma}_r \). Since \( \hat{\sigma}_{i+1} \) is inductively guaranteed to be an element of \( \hat{\Sigma}_r \), this
conditions is always satisfied.

\[ \exists X'_{K}. F(\hat{X}, \hat{X}') = F(\hat{X}, \hat{X}') \] where \( F(\hat{X}, \hat{X}') = R_\varphi(\hat{X}, \hat{X}') \land \hat{\sum}_r(\hat{X}') \) \hfill (5.3)

\[ \exists X'_s. F_s(\hat{X}, (X'_{ed} \cup X'_s)) = F_s(\hat{X}, (X'_{ed} \cup X'_s)) \] where \( F_s(\hat{X}, (X'_{ed} \cup X'_s)) = \forall X'_{ep}. \exists X'_{K}. F(\hat{X}, \hat{X}') \). \hfill (5.4)

In the first equation \( X'_{K} \) is the set of unknown variables and \( \hat{X} \cup (\hat{X}' \setminus X'_{K}) \) is the set of independent variables.

In the second equation \( X'_s \) is the set of unknown variables and \( \hat{X} \cup X'_{ed} \) is the set of independent variables.

The solution for the color variables \( X'_{K} \) depends on the past variables \( \hat{X} \) and the set of variables \( X'_{ed} \cup X'_s \cup X'_{ep} \). However, the system variables \( X'_s \) can only depend on past variables and the set of variables \( X'_{ed} \) as the variables in \( X'_{ep} \) are private to the environment and the system has to select a letter from \( \Sigma_s \) without knowing the environment’s choice of letter from \( \Sigma_{ep} \). This is the reason for the existential quantification of the color variables \( X'_{K} \) and the universal quantification of the environment’s private variables \( X'_{ep} \) in the second Boolean equation. As was the case previously, once the solution of the \( X'_s \) variables is computed, the \( X'_s \) variables are replaced by their solutions in the solutions for the \( X'_{K} \) variables, causing those solutions to depend only on \( X'_{ed} \cup X'_{ep} \). This is because the objective is to find a solution for the \( X'_{K} \) variables in terms of \( X'_{ed} \) and \( X'_{ep} \).

The existence of a solution to these Boolean equations has already been established by checking the realizability of \( \varphi \) through \( R_{in} \) and \( R_\varphi \). This will become apparent when the existence of a solution to a Boolean equation is discussed in the next section.

### 5.8 Boolean Equations

This section reviews the theory of Boolean equations [Bro90]. It has been shown in the previous section that the synthesis of safety properties is reduced to a combinational synthesis problem. The combinational synthesis problem can be solved using Boolean equations. This section describes how to obtain the parametric solution for a Boolean equation.

Let \( x_1, \ldots, x_m \) and \( y_1, \ldots, y_n \) be two sets of variables ranging over a Boolean algebra \( B \). A Boolean
equation in independent variables \( x_1, \ldots, x_m \) and unknowns \( y_1, \ldots, y_n \) is a formula of the form

\[
\forall x_1, \ldots, x_m \cdot \exists y_1, \ldots, y_n \cdot F_0(x_1, \ldots, x_m) = F(y_1, \ldots, y_n, x_1, \ldots, x_m), \tag{5.5}
\]

where \( F : B^{m+n} \to B \) is a Boolean function and \( F_0 = \exists y_1, \ldots, y_n . F \) is the consistency condition of \( F \). Since (5.5) is fully determined when \( F \) is given, when no confusion arises, one can write \( F \) to signify (5.5).

A particular solution of (5.5) is a tuple \((f_1, \ldots, f_n) \in B^n \) such that

\[
\forall x_1, \ldots, x_m . F_0(x_1, \ldots, x_m) = F(f_1, \ldots, f_n, x_1, \ldots, x_m).
\]

Oftentimes when solving equations, and in this chapter in particular, \( B \) is the free Boolean algebra generated by a set of variables. For instance, it may be that \( f_i = f_i(x_1, \ldots, x_m) \), for \( 1 \leq i \leq n \).

A general solution in parametric form of (5.5) is a set of Boolean functions \( g_i(x_1, \ldots, x_m, p_1, \ldots, p_i) \), for \( 1 \leq i \leq n \), where each \( p_j \) is a parameter ranging over \( B \), such that

\[
\forall p_1, \ldots, p_n . \forall x_1, \ldots, x_m . F_0(x_1, \ldots, x_m) = F(g_1, \ldots, g_n, x_1, \ldots, x_m),
\]

and for every particular solution \((f_1, \ldots, f_n)\) of (5.5) there is a choice of \( p_j \)'s that produces a particular solution \((f'_1, \ldots, f'_n)\) such that, for \( 1 \leq i \leq n \),

\[
\forall x_1, \ldots, x_m . F_0(x_1, \ldots, x_m) \leq f_i(x_1, \ldots, x_m) \leftrightarrow f'_i(x_1, \ldots, x_m).
\]

Example 5.8.1. As an example, \( F = y_1 \lor y_2 \lor \neg x_1 \), a general solution in parametric form for \( y_1 \) and \( y_2 \) is:

\[
F_0 = \exists y_2 . \exists y_1 . F = \top
\]

\[
g_1 = p_1
\]

\[
g_2 = (\neg p_1 \land x_1) \lor p_2.
\]

Let \( p_1 = \bot \) and \( p_2 = \bot \) then the particular solution \( f_1 = \bot \) and \( f_2 = x_1 \) is obtained. One can verify that \( g_1 \lor g_2 \lor \neg x_1 = \top \) regardless of the choice of \( p_1 \) and \( p_2 \).

A general solution to (5.5) can be computed by the method of successive eliminations [Rud74, Bro90], which, given \( F \), returns \( F_0 \) and the solution functions \( g_i \). Letting \( F_n = F \) and \( F_{i-1} = \exists y_i . F_i \)
for $1 \leq i \leq n$, it produces
\[
  g_i = \neg F_i(g_1, \ldots, g_{i-1}, \bot, x_1, \ldots, x_m) \lor (p_i \land F_i(g_1, \ldots, g_{i-1}, \top, x_1, \ldots, x_m)),
\]  
(5.6)
where $\neg$, $\lor$, and $\land$ to denote complementation, join, and meet in $B$, respectively. Note that (5.6) is independent of the Boolean algebra $B$. That is, general solutions can be computed before the algebra $B$ is known from which a particular solution must be chosen. The importance of this property will be discussed in the next section.

**Example 5.8.2.** Consider $F = (\neg x_1 \land y_1) \lor (x_2 \land y_2)$. Then the method of successive eliminations proceeds as follows:

\[
  F_2 = F = (\neg x_1 \land y_1) \lor (x_2 \land y_2) \\
  F_1 = \exists y_2 . F_2 = (\neg x_1 \land y_1) \lor x_2 \\
  F_0 = \exists y_1 . F_1 = \neg x_1 \lor x_2 \\
  g_1(x_1, x_2, p_1) = \neg x_2 \lor p_1 \\
  g_2(x_1, x_2, p_1, p_2) = x_1 \lor (x_2 \land \neg p_1) \lor p_2.
\]

One can verify that $\forall p_1, p_2 . \forall x_1, x_2 . \neg x_1 \lor x_2 = F(g_1, g_2, x_1, x_2)$.

Setting $p_1 = p_2 = \bot$ in $g_1$ and $g_2$, one obtains the particular solution $f_1 = \neg x_2$, $f_2 = x_1 \lor x_2$. The same solution is obtained for $p_1 = \neg x_2$ and $p_2 = x_1 \lor x_2$. The particular solution $f'_1 = \neg x_2$, $f'_2 = x_2$ cannot be obtained from $g_1$ and $g_2$ because $g_2 \geq x_1$, but $f'_2$ is not; however, for $i \in \{1, 2\}$,
\[
  \forall x_1, x_2 . \neg x_1 \lor x_2 \leq f_i \leftrightarrow f'_i.
\]

When the consistency condition is identically satisfied, it can be shown that
\[
  \neg F_i(g_1, \ldots, g_{i-1}, \bot, x_1, \ldots, x_m) \leq F_i(g_1, \ldots, g_{i-1}, \top, x_1, \ldots, x_m).
\]

Therefore, if $p_i$ is taken in the interval defined by the two bounds, the particular solution obtained for $y_i$ is $p_i$ itself.
A Boolean equation defines a two-player game in which one player selects a value \((\hat{x}_1, \ldots, \hat{x}_m) \in (\{\bot, \top\})^m\) for the independent variables, while the other must choose a value \((\hat{y}_1, \ldots, \hat{y}_n) \in (\{\bot, \top\})^n\) for the unknowns such that \(F_0(\hat{x}_1, \ldots, \hat{x}_m) = F(\hat{y}_1, \ldots, \hat{y}_n, \hat{x}_1, \ldots, \hat{x}_m)\). A particular solution to the equation gives one winning strategy for the second player, while a general solution describes all such winning strategies (that differ over the consistency condition).

The following definition describes the parameterized transducer that can be generated from the solution of (5.1), (5.2), (5.3), and (5.4).

**Definition 5.8.3.** Let \(\hat{W}\) be a fusion-closed language defined over the variables \(X_{ed} \cup X_s \cup X_{ep} \cup X_K \cup X'_{ed} \cup X_s \cup X'_{ep} \cup X'_K\) (where the language \(\hat{W}\) embeds the safety language described by the safety property \(\varphi\) which is defined over the variables \(X_{ed} \cup X_s \cup X_{ep}\)). Let \(P \cup X'_{ed} \cup X'_{ep}\) be the set of inputs where \(P\) is the set of parameters. Let \(X \cup I\) be the set of latches where the initial value of each latch is \(\bot\) while the next value of \(I\) is \(\top\) and the next value of remaining latches is defined by the corresponding signals in \(X'_{ed} \cup X_s \cup X'_{ep} \cup X'_K\). Let \(X'_s\) be the outputs and let \(X'_K\) be the set of internal signals where every variable \(x' \in X'_s \cup X'_K\) represents a function defined as:

\[
x' = (\neg I \land x'_{in}) \lor (I \land x'_{\infty}),
\]

where \(x'_{in}\) is the parametric solution obtained from the equations (5.1) and (5.2) while \(x'_{\infty}\) is the parametric solution obtained from equations (5.3) and (5.4). The collection of these inputs, outputs, latches, and internal signals is the parameterized transducer \(M_\varphi\).

Notice that in the definition of \(M_\varphi\) the inputs \(X'_{ed} \cup X'_{ep}\) correspond to the inputs \(X_{ed} \cup X_{ep}\) in \(\varphi\) while the output \(X'_s\) corresponds to the input \(X_s\) of \(\varphi\). It is shown in Section 5.9 that the parameterized transducer \(M_\varphi\) captures all the system’s strategies in guaranteeing the satisfaction of a safety property \(\varphi\). The following example describes the construction of an \(M_\varphi\) from a realizable fusion-closed and limit-closed language.

**Example 5.8.4.** Continuing Example 5.6.1, the synthesis of a transducer \(M_\varphi\) through the synthesis of \(x', y'\) and \(r'\) is shown here; \(\{x', y'\}\) is the set of color variables while \(\{r'\}\) is the set of system variables. From
(5.3), the steady state Boolean equation for the unknown variables \( \{x', y'\} \) is \( \exists\{x', y'\} \cdot F = F \) where \( F = R_\phi(\hat{X}, \hat{X}^t) \lor \Sigma^\phi_r(\hat{X}^t) \), where \( \{r', g', x, y, g, r\} \) are the independent variables. (Only the variables \( \{g, g'\} \) are truly independent variable while the rest are controlled by the system. Therefore, \( \Sigma^\phi_r \neq \emptyset \) guarantees that the equation \( \exists\{x', y'\} \cdot F = F \) has a non-empty consistency condition.) The relation \( R_\phi \) and the alphabet \( \Sigma^\phi_r \) have been computed in Example 5.6.1. Let \( x_i \) and \( y_i \) be two parameters, then the steady state solution of variables \( \{x', y'\} \) is given by:

\[
y'_\infty = \neg(F_2) \lor (y_i \land (F_2)) = r \land \neg g \land \neg r' \land \neg g',
\]
\[
x'_\infty = \neg F_{-x'} \lor (x_i \land F_{x'}) = \neg r \lor g \lor x \lor g',
\]

where

\[
F_2 = \exists x' \cdot F = ((r \land \neg g \land \neg x \land (r' \lor g')) \lor (x \land \neg y') \lor (x \land \neg r' \land \neg g')) \land
\]
\[
((r \land \neg g \land \neg y \land (r' \lor g') \land \neg y') \lor
\]
\[
((-r \lor g) \land \neg y \land \neg y') \lor
\]
\[
(r \land \neg g \land \neg y \land \neg r' \land \neg g' \land y') \lor
\]
\[
((-r \land \neg g \land y \land (r' \lor g') \land \neg y')).
\]

The steady state solution of the variables in \( \{r'\} \) is computed from the Boolean equation \( \exists r' \cdot F_s = F_s \) where \( F_s(\hat{X}, (X_{ed}', X_s')) = \forall g' \cdot \exists\{x', y'\} \cdot F(\hat{X}, \hat{X}') \) (as described by (5.4)).

\[
\exists\{x', y'\} \cdot F = \exists y' \cdot F_2 = ((r \land \neg g \land \neg y \land (r' \lor g')) \lor ((\neg r \lor g) \land x \land \neg y) \lor
\]
\[
(-r \land \neg g \land x \land y \land (r' \lor g')) \lor (r \land \neg g \land \neg y \land \neg r' \land \neg g')\)
\]
\[
F_s = \forall g' \cdot \exists y' \cdot F_2 = ((r \land \neg g) \land \neg r') \land (\neg y \lor (\neg r \land \neg g \land r')) \land
\]
\[
F_0 = \exists r' \cdot F_s = (x \land \neg y) \lor (x \land \neg r \land \neg g) \lor (\neg y \land r \land \neg g).
\]

(Since \( X_{ed} = \emptyset \), it should come as no surprise that \( F_0(\hat{X}) = \Sigma^\phi_r(X) \), which implies that the solution of \( r' \) is valid when the previous letter belongs to \( \Sigma^\phi_r \).) Let \( r_i \) be another parameter then the steady state solution of \( r' \) is:

\[
r'_\infty = \neg(F_s) \land r_i \land (F_s) \land r_i = r_i \lor r \land x \land y.
\]
The value of \( r' \) is replaced with its solution in the solution of \( x'_{\infty} \) and \( y'_{\infty} \) which yields the following solution:

\[
\begin{align*}
y'_{\infty} &= r \land \neg g \land \neg r_i \land x \land \neg y \land \neg g' , \\
x'_{\infty} &= \neg r \lor g \lor x \lor g' ,
\end{align*}
\]

Likewise the initial values for \( \{ x', y' \} \) are synthesized from \( \exists \{ x', y' \} . I(\hat{X}') = I(\hat{X}') \) where \( I(\hat{X}') = R_{in}(\hat{X}') \land \Sigma_{\hat{X}'}(\hat{X}') \). (The predicate \( \Sigma_{\hat{X}'} \) and the alphabet \( \Sigma_{\hat{X}'} \) have been computed in Example 5.6.1.)

\[
\begin{align*}
y'_{in} &= (I_2)_y \lor (y_i \land (I_2)_{y'}) = \bot , \\
x'_{in} &= \neg I_{-x'} \lor (x_i \land I_{x'}) = g' , \quad \text{where} \\
I_2 &= \exists x'. I = (r' \lor g') \land \neg y' .
\end{align*}
\]

The initial value of \( r' \) is computed from \( \exists r'. I_s = I_s \) where \( I_s(X'_{ed} \cup X'_s) = \forall X'_e . \exists X'_K . I(\hat{X}') \):

\[
\begin{align*}
\exists \{ x', y' \} . I &= (r' \lor g') \\
I_s &= \forall g' . \exists \{ x', y' \} . I = r' , \quad I_0 = \exists r'. I_s = \top , \\
r'_{in} &= (I_s)_{-r'} \lor ((r_i \land (I_s)r') = \top .
\end{align*}
\]

Each variable \( v \in \hat{X} \) represents a latch which stores the current value of the corresponding value of \( v' \in \hat{X}' \). Each variable \( o' \in X'_s \) represents the output of the transducer and is labeled as the corresponding variable \( o \). Each variable \( o' \in X'_K \) is stored in the latches represented by the variables in \( X_K \). These are treated as internal signals (as these signals are not the outputs specified in the specification).

An \( M_\varphi \) can now be constructed where \( \{ r_i, x_i, y_i, g' \} \) are the inputs, \( \{ r, x, y, g, I \} \) are the latches (the initialization and next values of latches have been described in Section 5.7), \( \{ r' \} \) is the output, and \( \{ x', y' \} \) are the internal signals.

The solution is kept in parameterized form so that winning strategies for the progress properties may be chosen from the set of all strategies that win the safety game. This is done by computing the appropriate values of the parameters (which may be functions requiring some finite memory to satisfy the progress properties). If the specification does not contain any progress properties then an implementation \( M_\varphi \) without any parameters can be obtained by assigning any values to the parameters.
5.9 Correctness and Importance of Parameterized Solutions

Chapter 4 describes in detail how a specification that is the conjunction of several properties can be synthesized incrementally. In particular, when the specification contains a safety property, then one can always compute the winning strategies for the safety game up front. In short, let $G_S$ be the game with conjunctive parity winning condition obtained from the specification and let $G_s$ be the game obtained from some safety property in the specification. Let $T$ be the set of winning strategies computed after playing the game $G_s$; then the system’s winning strategies for $G_S$ must be contained in $T$. The focus of this chapter has been the development of an alternative to the automata-based approach of Chapter 4 to compute the winning strategies of safety games such that their symbolic representation is significantly smaller.

Given a safety property that must by satisfied by the system, one can first derive an $\mathcal{R}$-generable language and initial predicate that embed the safety property as discussed in Section 5.4. One can then obtain a parameterized representation of a transition function that guarantees the satisfaction of the safety property as discussed in Sections 5.7 and 5.8. The choice of a particular strategy to win the safety game amounts to the choice of the values of the parameters in the parameterized transition function. The game corresponding to the progress properties is therefore played on a graph defined by this parameterized transition function and the automata for the progress properties. In so doing, every strategy for the progress properties is guaranteed to also be a strategy for the safety property. In this game, the system controls the parameters, which in turn define its response to the environment’s moves. The parameters are chosen from a Boolean algebra that may be larger than that generated by the environment’s variables because it may include the state variables required by, say, conjunctive parity games.

This section provides a proof that the parameterized representation is in fact a representation of all the strategies of a safety game obtained from $\varphi$. The intuitive idea behind this new representation is illustrated in the following examples. The first one shows that the values of the parameters cannot be restricted to $\bot$ and $\top$.

**Example 5.9.1.** Let the specification be the conjunction of two properties $\varphi_1$ and $\varphi_2$, where $\varphi_1 = G(\neg h \rightarrow X\neg x)$ is a safety property and $\varphi_2 = GF(h \rightarrow X(a \leftrightarrow x))$ is a recurrence property. Let $X_{ed} = \{a\}$,
Figure 5.12: The parameterized transducer $M_{\varphi_1}$ that captures all the strategies to satisfy $\varphi_1$. The output function $x$ is $q \land x_i$.

$X_s = \{x\}, X_{ep} = \emptyset$, and the set of parameters $P = \{x_i\}$; then the parameterized representation $M_{\varphi_1}$ satisfies $\varphi_1$. The parameterized representation $M_{\varphi_1}$ is shown in Figure 5.12. It is apparent from the graph that strategies for $\varphi_1$ require $h$ to be remembered. However, $x$ is constrained only when preceded by $\neg h$.

The parameter $x_i$ in $x = q \land x_i$ allows the system to choose its response in the other case.

If the parameterized representation $M_{\varphi_1}$ is simplified by fixing the value of the parameter $x_i$ to either $\bot$ or $\top$, then $M_{\varphi_1}$ cannot satisfy $\varphi_2$ because the environment can always keep asserting $h$ infinitely often and assigning $a = \neg x_i$ indefinitely. Hence the consequent of $\varphi_2$ can never be satisfied.

However, $M_{\varphi_1}$ can satisfy $\varphi_2$ when $x_i$ is made a function of $a$. Since $x = q \land x_i$, one possible strategy chooses $x_i = a$, which implies that $x = a$ when $h$ held at the previous step. With this strategy the system is able to satisfy the recurrence property $\varphi_2$.

Example 5.9.2. Let the specification be the conjunction of three properties $\varphi_1$, $\varphi_2$, and $\varphi_3$, where $\varphi_1 = \text{G}(a \land b \rightarrow x)$ is a safety property and $\varphi_2 = \text{GF}(a \leftrightarrow x)$ and $\varphi_3 = \text{GF}(b \leftrightarrow x)$ are recurrence properties. Let $X_{ed} = \{a, b\}$, $X_s = \{x\}$, $X_{ep} = \emptyset$, and the set of parameters $P = \{x_i\}$; then the parameterized representation $M_{\varphi_1}$ is $x = (a \land b) \lor x_i$; this representation satisfies $\varphi_1$ for every choice of $x_i$.

If the environment infinitely often chooses input letter $a \land b$ then the properties $\varphi_2$ and $\varphi_3$ are automatically satisfied. If the environment infinitely often chooses the letter $\neg a \land \neg b$ then the system can satisfy $\varphi_2$ and $\varphi_3$ by de-asserting $x_i$. However, if the environment chooses these two letters only a finite number of times, then the system can only satisfy $\varphi_2$ by setting $x_i$ to match $a$ from time to time. Likewise for $\varphi_3$ and $b$. The system’s strategy needs memory. In this specification a single bit of memory is sufficient to remember which of $a$ and $b$ was last matched. Let this bit be $y$. Then $y' = \neg y$ and $x_i = (y \land a) \lor (\neg y \land b)$ provide a
winning strategy.

As it is shown in Example 5.9.2, the winning strategy for the whole specification may require memory to be added. Thus the winning strategy may require the parameters to be functions of current variables and additional memory variables. Fortunately, this is not a problem with the parameterized representation of the transition function because the parameterized solutions are independent of the Boolean algebra \( B \).

It is now shown that if there is no winning strategy for the progress properties within the allowed behaviors of the parameterized representation then the specification is unrealizable and vice versa. Let the parameterized transducer be \( M_\varphi \) (Definition 5.8.3) with inputs \( X_{ed} \cup X_{ep} \cup P \) where \( P = \{ p_1, p_2, \ldots, p_n \} \) be the set of parameters and the outputs \( X_s \cup X_K \) (the internal signals \( \Sigma_K \) are treated as outputs for the purpose of the proof of Lemma 5.9.3). The input alphabet of the transducer \( M_\varphi \) is \( \Sigma_{ed} \times \Sigma_{ep} \times \Sigma_P \). The output alphabet is \( \Sigma_s \times \Sigma_K \). The language of \( M_\varphi \) is over the alphabet \( \Sigma_{ed} \times \Sigma_{ep} \times \Sigma_P \times \Sigma_{s} \times \Sigma_K \) (it has been discussed in Section 2.1 that the language of a transducer is defined over its input and output alphabet). Let \( \hat{\mathcal{G}}^R_{\varphi} \) be the restriction of \( \hat{\mathcal{G}}^R_{\varphi} \) to \( \hat{\Sigma}_{\varphi}^R \) such that all the initial letters of this game belong to \( \Sigma_{\varphi}^R \) and the relation of this game is a subset of \( \Sigma_{\varphi}^R \times \Sigma_{\varphi}^R \) such that if \( \sigma \in \Sigma_{\varphi}^R \), \( \sigma' \in \Sigma_{\varphi}^R \) and \( R_{\varphi}(\sigma, \sigma') \) is satisfied then such a pair of letters belongs the relation of \( \hat{\mathcal{G}}^R_{\varphi} \). Let the projection function \( \Gamma \) be defined as follows:

\[
\Gamma : \Sigma_{ed} \times \Sigma_{ep} \times \Sigma_P \times \Sigma_s \times \Sigma_K \rightarrow \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep} \times \Sigma_K.
\]

Then the function \( \Gamma(A) \) is written for the point-wise extension of \( \Gamma \) to set of letters \( A \), \( \Gamma(w) \) for the point-wise extension of \( \Gamma \) to a word \( w \in \hat{\Sigma}^\omega \) and \( \Gamma(W) \) for the point-wise extension of \( \Gamma \) to set of words \( W \). The language of \( M_\varphi \) is connected to the language of \( \hat{\mathcal{G}}^R_{\varphi} \) through the \( \Gamma \) function which is proven in the following lemma.

**Lemma 5.9.3.** Given \( M_\varphi \) and \( \hat{\mathcal{G}}^R_{\varphi} \), then \( \Gamma(L(M_\varphi)) = L(\hat{\mathcal{G}}^R_{\varphi}) \).

**Proof.** Suppose \( L(\hat{\mathcal{G}}^R_{\varphi}) \setminus \Gamma(L(M_\varphi)) \neq \emptyset \) then there exists a word \( \hat{w}_0 \hat{w}_1 \ldots \hat{w}_i \hat{w}_{i+1} \ldots \in L(\hat{\mathcal{G}}^R_{\varphi}) \) such that \( \hat{w}_0 \hat{w}_1 \ldots \hat{w}_i \hat{w}_{i+1} \ldots \notin \Gamma(L(M_\varphi)) \). (Every letter \( \hat{w}_i \) is a tuple of letters \( (\sigma_{ed}^i, \sigma_s^i, \sigma_{ep}^i, \sigma_K^i) \).) Notice that \( \hat{w}_0 = (\sigma_{ed}^0, \sigma_s^0, \sigma_{ep}^0, \sigma_K^0) \) belongs to the set \( R_{in} \cap \Sigma_{\varphi}^R \) and therefore it satisfies the right hand side of the equation (5.1) and \( (\sigma_{ed}, \sigma_s) \) satisfies the right hand side of the equation (5.2). This implies that there
exists a word in $L(M_\varphi)$ such that the $\Gamma$ projection of the initial letter of this word is $\hat{w}_0$. Furthermore, notice that for every $i \geq 0$, $R_\varphi(\hat{w}_i, \hat{w}_{i+1})$ is satisfied; then any two consecutive pair of letters $\hat{w}_i$ and $\hat{w}_{i+1} = (\sigma_e^{i+1}, \sigma_s^{i+1}, \sigma_K^{i+1})$ satisfy the right hand side of equation (5.3) while $\hat{w}_i$ and $(\sigma_e^{i+1}, \sigma_s^{i+1})$ satisfy the right hand side of the equation (5.4). Therefore, if the projection of the current letter generated by $M_\varphi$ is the letter $\hat{w}_i$ then there exists a way for $M_\varphi$ to generate a new letter such that its $\Gamma$ projection is $\hat{w}_{i+1}$. Furthermore, let the prefix $\hat{w}_0 \hat{w}_1 \ldots \hat{w}_i$ and there exists another word $\hat{w}_0' \hat{w}_1' \ldots \hat{w}_i' \ldots \in L(M_\varphi)$ such that $\hat{w}_0 \hat{w}_1 \ldots \hat{w}_i = \Gamma(\hat{w}_0' \hat{w}_1' \ldots \hat{w}_i')$ then there exists an assignment to the parameters of $M_\varphi$ such that $\hat{w}_0 \hat{w}_1 \ldots \hat{w}_i \hat{w}_{i+1} = \Gamma(\hat{w}_0' \hat{w}_1' \ldots \hat{w}_{i} \hat{w}_{i+1})$ (and there exists a way to extend the prefix $\hat{w}_0' \hat{w}_1' \ldots \hat{w}_{i} \hat{w}_{i+1}$ to a word in $L(M_\varphi)$). This contradicts the assumption hence $L(\hat{G}_\varphi^R) \subseteq \Gamma(L(M_\varphi))$.

Suppose $\Gamma(L(M_\varphi)) \setminus L(\hat{G}_\varphi^R) \neq \emptyset$ then there exists a word $\hat{w}_0' \hat{w}_1' \ldots \hat{w}_i' \hat{w}_{i+1}' \ldots \in L(M_\varphi)$ such that $\Gamma(\hat{w}_0' \hat{w}_1' \ldots \hat{w}_i' \hat{w}_{i+1}') \notin L(\hat{G}_\varphi^R)$. From (5.1) and (5.2) one notices that the initial letter $\hat{w}_0'$ is such that $\Gamma(\hat{w}_0') \in R_{in} \cap \Sigma_\varphi^e$. From (5.3) and (5.4) one notices that for every assignment to parameters of $M_\varphi$ and for every $i \geq 0$ the relation $R_\varphi(\Gamma(\hat{w}_i), \Gamma(\hat{w}_{i+1}))$ is satisfied. This implies that $\Gamma(\hat{w}_0' \hat{w}_1' \ldots \hat{w}_i' \hat{w}_{i+1}') \in L(\hat{G}_\varphi^R)$, which contradicts the assumption. Therefore $\Gamma(L(M_\varphi)) \subseteq L(\hat{G}_\varphi^R)$.

Hence it has been shown that $\Gamma(L(M_\varphi)) = L(\hat{G}_\varphi^R)$. \hfill \Box

The following example is a continuation of Example 5.6.1. This example illustrates how through the assignments of the parameters, a run of $M_\varphi$ can match any run of $G_\varphi$.

Example 5.9.4. Consider the property $\varphi = \varphi_1 \land \varphi_2$ over the alphabet $2^{\{r,g\}}$, where $\varphi_1 = r \land g$ and $\varphi_2 = G(r \land \neg g \rightarrow X(r \lor g \lor X(r \lor g)))$. If $X_{ed} = \emptyset$, $X_s = \{r\}$, and $X_{ep} = \{g\}$, the property $\varphi$ has been proven to be realizable in Example 5.6.1.

The input-based game $G_\varphi$ is derived from an irredundant automaton for $\varphi$. The game $G_\varphi$ shown in Figure 5.13 is the restriction of $G_\varphi$ to the system’s winning states and strategies, while the game $G_\varphi$ is shown on the left in Figure 5.11. The parameterized $M_\varphi$ is described in Example 5.8.4.

Table 5.2 shows the different ways the system can complete the letter to produce an accepting run of $G_\varphi$ for the input sequence $-g, g, -g, -g, g$. The second column shows that through the parameters the system can simulate $G_\varphi$ for the same input sequence. For instance the sequence $rrr-rr$ can be generated
Figure 5.13: The game $G_\varphi$ obtained from the property $\varphi = \varphi_1 \land \varphi_2$, where $\varphi_1 = r \rightarrow W g$ and $\varphi_2 = G(r \land \neg g \rightarrow X(r \land g \lor X(r \land g)))$. In this game the environment moves after the system makes its move. The game $G_\varphi$ is derived from $G_{\varphi}$ by restricting it to system’s winning strategies.

by $M_\varphi$ by either of these sequences

\[ r_i r_i \neg r_i \neg r_i \neg r_i , \quad r_i \neg r_i \neg r_i \neg r_i \neg r_i , \]
\[ \neg r_i \neg r_i \neg r_i \neg r_i \neg r_i , \quad \neg r_i \neg r_i \neg r_i \neg r_i \neg r_i . \]

One can verify that $M_\varphi$ can simulate $G_\varphi$ by selecting appropriate values of the parameters for any input sequence. Similarly, $G_\varphi$ can simulate $M_\varphi$ irrespective of the values of the parameters.

Table 5.2: When the environment chooses to play the prefix $\neg g, g, \neg g, \neg g, g$ of then the possible runs of $G_\varphi$ and the corresponding runs of $M_\varphi$ are shown here. The label $S_i$ denotes the state of $M_\varphi$ in the $i^{th}$ turn, while $P_i$ denotes the value of the parameters in the $i^{th}$ turn, and $X_{s_i}$ denotes the value of the output variable in the $i^{th}$ turn.
5.10 Obligation Properties and Transition Constraints

If a game with an $\omega$-regular winning condition has a graph with more than one strongly connected component (SCC) then the winning and losing states can be computed inductively starting from the terminal SCCs. At each non-terminal SCC, one computes the states that each player can control to its winning states outside of the SCC (which are already known). The game is then played on the remaining states. This approach is discussed in [LF09]. In this section this idea is applied to the obligation properties defined by the implication of two safety properties (e.g., environment assumption and system guarantee). Every obligation property which is not a safety property results in a DPW of minimum index 1 with more than one SCC; this observation allows one to solve the game (from the point of view of the system) obtained from an obligation property which is an implication between two safety properties through the relation-based approach described in this chapter.

To check realizability of an implication between two safety properties such as $\psi \rightarrow \varphi$, one converts $\psi$ and $\varphi$ to parity games $G_\psi$ and $G_\varphi$ with safety conditions $\pi_\psi$ and $\pi_\varphi$. The SCCs of their product can be partitioned in three sets; the bottom set $S_B$ contains the states in which the antecedent has been violated. The middle set $S_M$ contains the states in which only the consequent has been violated, and the top set $S_T$ contains the states where both properties hold. The states in $S_T \cup S_B$ have priority 1; those in $S_M$ have priority 0. This structure is illustrated in the following example.

Example 5.10.1. Consider the LTL formula $\Phi = \psi \rightarrow \varphi$, where $\psi = G(r \lor g)$ and $\varphi = G(r \land g \rightarrow X g)$. The property states that as long as $r$ and $g$ are never de-asserted simultaneously then $g$ should not be de-asserted after both $r$ and $g$ are asserted.

The input-based game $G_\Phi$ is shown in Figure 5.14, where $X_{ed} = \emptyset$, $X_s = \{g\}$, and $X_{ep} = \{r\}$. The game has three SCCs where $S_T = \{q_{in}, q_1\}$, $S_M = \{q_2\}$, and $S_B = \{q_3\}$. The protagonist wins the game by satisfying $\varphi$ as it cannot falsify $\psi$.

The games obtained from properties which are implication between two safety properties do not need to be built explicitly. Given an implication $\Phi = \psi \rightarrow \varphi$, where both $\psi$ and $\varphi$ are safety properties, one can obtain the relations $R_\psi$ and $R_\varphi$ as described in Sec. 5.4. These relations are used to check the realizability
of $\Phi$. First the subset of the alphabet is computed from which the environment is able to satisfy $\psi$ and then from within this subset the set of letters is computed from which the system can satisfy $\varphi$ as long as the environment keeps satisfying $\psi$. The pre-image operator $\mathrm{MX}_\varphi$ defined in Sec. 5.5 cannot be used for checking $\Phi$ because it computes the states that can be forced by the protagonist to stay within the SCC, while in a game obtained from $\Phi$, the protagonist may be able to win the game by staying within $S_T$ or by forcing a move out of $S_T$ to states from which it can force the play to $S_B$. Therefore, a modified pre-image operator needs to be defined that takes into account the protagonist’s option to escape the SCC $S_T$.

The solution of the game obtained from $\Phi$ follows three steps. In the first step one plays the game $\mathrm{MG}_\psi \top$ to compute the letters from which the environment can satisfy $R_\psi$. In this game the environment is the protagonist and the system is the antagonist. One may need to augment the alphabet $\Sigma$ to $\hat{\Sigma} = \Sigma \times K_e$ as described in Sec. 5.4; the control of coloring variables $X_{K_e}$ is assigned to the environment. The system is eventually able to force a violation of $R_\psi$ from the letters in $\hat{\Sigma} \setminus \Sigma_\psi^\top$. From the letters $\Sigma_\psi^\top$, the system can only satisfy $R_\varphi$ by satisfying $R_\psi$. The new pre-image operator is used to compute $\mathrm{MG}_\Phi \top$; the objective of the system is to keep satisfying $R_\varphi$ while the environment cannot use strategies that will give the system the option to choose the letter from $\hat{\Sigma} \setminus \Sigma_\varphi^\psi$. Once again, one may need to augment the alphabet $\hat{\Sigma}$ to $\hat{\Sigma} = \hat{\Sigma} \times K_s$. The system is able to satisfy $R_\Phi$ from the letters in $\Sigma_\varphi^\Phi = \mathrm{MG}_\Phi \top \cup (\hat{\Sigma} \setminus (\Sigma_\varphi^\psi \times K_s))$. Finally, the system wins the game obtained from $\Phi$ when the constraint $R_\text{in}^\psi \rightarrow R_\text{in}^\varphi$ allows the system to select a letter from $\Sigma_\varphi^\Phi$.

The details of the pre-image operator $\mathrm{MG}_\Phi$ are now discussed. Let the set of variables appearing in
the obligation property $\Phi = \psi \rightarrow \varphi$ be $X_\Phi = X_{ed} \cup X_s \cup X_{ep}$. The initial predicate $R_{in}^\psi$ and relation $R_\psi$ are obtained from $\psi$, where $R_\psi$ is a relation over the variables $X_\Phi \cup X_{K_e}$. (The variables in $X_{K_e}$ are the coloring variables used in $R_\psi$.) The first step is to check if the environment can win the safety game $\hat{G}_R^\psi$; this is done by checking if $MX_{in}^\psi MG_\psi T = T$. (The quantifiers in $MX_{in}^\psi$ and $MX_\psi$ are inverted to account for the fact that the environment is the one trying to win the game by satisfying $\psi$.) When it is the system trying to satisfy the property $\varphi$ then the $MX_\varphi T$ is defined as follows:

$$MX_\varphi T = \forall X'_{ed} . \exists X'_s . \forall X'_{ep} . \exists X'_{K_e} . R_\varphi(X, X') \land T(X') .$$

When it is the environment trying to satisfy the property $\varphi$ then the $MX_\varphi T$ is defined as follows:

$$MX_\varphi T = \exists X'_{ed} . \forall X'_s . \exists X'_{ep} . \exists X'_{K_e} . R_\varphi(X, X') \land T(X') .$$

In the next step, the initial predicate $R_{in}^\varphi$ and relation $R_\varphi$ are obtained from $\varphi$, where $R_\varphi$ is a relation over the variables $X_\varphi \cup X_{K_e}$. (The variables in $X_{K_e}$ are the coloring variables used in $R_\varphi$. The coloring variables $X_{K_e}$ are only needed for the satisfaction of $R_\psi$ and do not contribute to the satisfaction of $R_\varphi$.) Let $\hat{X}_\varphi = X_\varphi \cup X_{K_e} \cup X_{K_s}$ and $\hat{X}'_\varphi$ be the next state versions of the variables in $\hat{X}_\varphi$. The game $G_{\hat{\psi}}^R$ is obtained from the game $\hat{G}_R^\psi$ such that the environment is only allowed to play according to its winning strategies in $\hat{G}_R^\psi$. Given $R_\varphi(\hat{X}, \hat{X}')$ the pre-image operator $MX_\varphi^L T(\hat{X}')$ is defined as follows:

$$\forall X'_{ed} . \exists X'_s . \forall X'_{ep} . \exists X'_{K_e} . \forall X'_{K_s} . (R_\varphi \land T) \lor L .$$

where $L(\hat{X}, \hat{X}')$ is defined as $-(R_\psi(\hat{X}, \hat{X}') \land \Sigma_\psi^\psi(\hat{X}'))$ and $\Sigma_\psi^\psi = MG_\psi T$. The quantification order of $X'_{K_s}$ and $X'_{K_e}$ does not matter because the variables in $X'_{K_s}$ only appear in $R_\varphi$ and the variables in $X'_{K_e}$ only appear in $L$ as a result of $R_\psi$.

The final step is to check if the system can satisfy its initial constraints in the presence of the environment’s initial constraint. The pre-image operator $MX_{in} T$ is defined as follows:

$$\forall X_{ed} . \exists X_s . \forall X_{ep} . \exists X_{K_s} . \forall X_{K_e} . (R_{in}^\varphi \land T) \lor \neg R_{in}^\varphi .$$

The system wins the game if $MX_{in}(\Sigma_\varphi^\psi(X)) = T$. A system’s winning strategy is to either reach a position from where it can force a violation of the environment’s constraint (assumption) or to choose $\sigma_s$ so that its constraint is satisfied (guarantee).
As discussed in Sec. 5.4, the control of coloring variables is assigned to the player who is trying to satisfy the property. This is why the variables in $X_{K_e}$ ($X_{K_s}$) are controlled by the environment (system) when it is trying to satisfy $\psi$ ($\varphi$). This distinction is at work in the following example which has been inspired by the AMBA bus controller [AL].

**Example 5.10.2.** Consider the LTL formula $\Phi = \psi \rightarrow \varphi$, where

\[
\psi = G(r \land Xr \rightarrow XX(r \rightarrow l)) \quad \text{and} \quad \varphi = G(r \land \neg l \rightarrow \neg g) \land G(r \land Xr \rightarrow XX(r \rightarrow g)) .
\]

Then

\[
R_{in}^\psi = \neg x \\
R_{\psi} = ((\neg r \rightarrow \neg x') \land (r \land \neg x \rightarrow ((\neg r' \land \neg x') \lor (r' \land x'))) \land \\
(r \land x \rightarrow ((r' \land l' \land x') \lor (\neg r' \land \neg x'))) \\
R_{in}^\varphi = (\neg r \lor l \lor \neg g) \land \neg y \\
R_{\varphi} = ((r \land \neg l \rightarrow \neg g) \land (\neg r \rightarrow \neg y') \land \\
(r \land \neg y \rightarrow ((\neg r' \land \neg y') \lor (r' \land y')) \land (r \land y \rightarrow ((r' \land g' \land y') \lor (\neg r' \land \neg y'))) .
\]

The alphabet is $\Sigma_\Phi = 2^{\{r,l,g\}}$ and the augmented alphabet is $\hat{\Sigma}_\Phi = 2^{\{r,l,g,x,y\}}$, where

\[
X_{ed} = \{r,l\} \quad X_s = \{g\} \quad X_{ep} = 0 \quad X_{K_e} = \{x\} \quad X_{K_s} = \{y\} .
\]

The system cannot win either games $G_{\neg \psi}$ or $G_{\varphi}$, but it can win the game $G_\Phi$. In the game $G_\varphi$ the environment can force the system to violate $\varphi$ at any time by playing the sequence $r \land \neg l, r \land \neg l, r \land \neg l$. On the other hand, in the game $G_\Phi$, this sequence forces the environment to violate $\psi$ and if the environment never plays this sequence then system can always satisfy $\varphi$. Thus $G_\Phi$ is won by the system.

In this section it has been shown that one can efficiently check the realizability of a special case of an obligation property; the obligation property which is an implication between two safety properties without explicitly constructing the game. The synthesis of a parametric representation which satisfies such a
property follows the same principles discussed in Section 5.7. The parametric representation only captures
the strategies of those winning states of $G_\Phi$ in which the environment does not violate $\psi$ and the system
satisfies $\varphi$. The reasons for excluding the states of $G_\Phi$ where the environment is forced to violate $\psi$ have
been discussed in development of the hierarchy based approach in Section 4.3. Section 5.11 discusses the
relevancy of this section in the context of specifications of reactive systems.

5.11 Relation-Based approach and common Specifications

This section discusses practically relevant examples of specifications of reactive systems. The hierar-
chy based approach of Chapter 4 can be used to synthesize the specifications described in this section. The
hierarchy based approach uses automata to synthesize both safety and progress properties in the specifica-
tion. However one can use relation-based approach described in this chapter to synthesize the safety and
obligation properties present in the specification. It is now briefly described how the relation-based approach
can be employed during the synthesis of various types of specifications.

Given the sets of variables $X_{ed}$ and $X_{ep}$ controlled by the environment and the set of variables $X_s$
controlled by the system where $X = X_{ed} \cup X_s \cup X_{ep}$, let $S_e^a$ and $S_e^b$ be any two sets of environment’s safety
properties and $L_e^a$ and $L_e^b$ be any two sets of environment’s progress properties defined over the variables in
$X$. Similarly, let $S_s^a$ and $S_s^b$ be any two sets of system’s safety properties and $L_s^a$ and $L_s^b$ be any two sets of
system’s progress properties defined over the variables in $X$. Then one can describe various types reactive
systems by these sets of properties.

1. $S_s^a \land L_s^a$: When the environment is unconstrained then the specification is of this form. A parame-
terized transducer (which can also be viewed as a parameterized transition function) that guarantees
the satisfaction of $S_s^a$ can be used to find the winning strategies for $L_s^a$. The transducer is composed
with a game obtained from $L_s^a$ using the hierarchy based approach, the system is allowed to control
the parameters of the transducer in search for a strategy to guarantee the satisfaction of $S_s^a \land L_s^a$.

2. $\langle S_e^a \rightarrow S_s^a \rangle \land \langle L_e^a \rightarrow L_s^a \rangle$ (or $\langle S_e^a \rightarrow S_s^a \rangle \land \langle L_e^a \rightarrow L_s^a \rangle \land \langle L_e^b \rightarrow L_s^b \rangle$): When $L_s^a$ and $L_s^b$ are
sets of recurrence properties then this specification is of $GR(1)$ type. (Similarly, if $L_e^a, L_e^b, L_s^a,$ and
...
$L^b_e$ are sets of recurrence properties then it is a specification of GR(2) type. An example of such specification has been discussed in Chapter 7.) The first step is to check if the system can negate $S^a_e$; if it can then the system has a strategy to guarantee the satisfaction of the specification by negating $S^a_e$. Otherwise $S^a_e \rightarrow S^a_s$ is an implication between two safety properties; the synthesis of this type of property through the relation-based approach has been discussed in Section 5.10. The parametric representation of the transition function that guarantees the satisfaction of $S^a_e \rightarrow S^a_s$ is then used to find the winning strategies for the progress property $L^a_e \rightarrow L^a_s$ (or $L^a_e \rightarrow L^a_s \land L^b_e \rightarrow L^b_s$) which has been discussed in more detail in Chapter 4.

(3) $S^a_e \land L^a_e \rightarrow S^a_s \land L^a_s$: When $L^a_e$ and $L^a_s$ are sets of recurrence properties then this specification is of GR(1) type. The first step is to check if the system can negate $S^a_e \land L^a_e$; if it can then the environment is not well-separated (the concept of a well-separated environment is discussed in Section 4.5, most practical examples require the environment to be well-separated) and the system has a strategy to satisfy the specification by negating $S^a_e \land L^a_e$. (In some cases, the system has a strategy to negate $S^a_e \land L^a_e$ if the environment makes a mistake; let these strategies be $T_a$.)

In the first step, the set of properties $S^a_e \land L^a_e$ are converted to a game by first obtaining a parametric representation for $S^a_e$ (in this step the environment is the protagonist) and then composing it with the game obtained from $L^a_e$. The composed game $G_e$ is restricted to environment’s winning states denoted as $G_e$; then a transition constraint $R_e$ is obtained from $G_e$ (it is as if the priority functions of $G_e$ have been replaced with another priority function which assigns the priority 1 to all the states of this game).

Thanks to Lemma 4.3.5, one does not need to play the game $G_e \times G_{S^a_s}$ and instead one can obtain the parametric representation $P^a_s$ that guarantees satisfaction of $R_e \rightarrow S^a_s$. This representation contains all the winning strategies for the game $G_e \times G_{S^a_s}$. This parametric representation is then composed with $G_e$ and the game obtained from $L^a_s$ and let $T_c$ be the set of strategies for the final game.

In the final step, an implementation is generated from the two strategies $T_a$ and $T_c$. 
(4) \((S^a_e \land L^a_e \rightarrow S^a_s \land L^a_s) \land (S^b_e \land L^b_e \rightarrow S^b_s \land L^b_s)\): This is an example of a GR(2) specification where the environment may not be well-separated. Establishing the realizability of such a specification requires that an implementation be generated as a proof that can guarantee the satisfaction of this specification whether by the system’s strategy that forces the violation of environment assumptions or guarantees the satisfaction of system guarantees.

This specification is treated as two separate GR(1) specifications and one game each for \((S^a_e \land L^a_e \rightarrow S^a_s \land L^a_s)\) and \((S^b_e \land L^b_e \rightarrow S^b_s \land L^b_s)\) are generated. Each game is then restricted to system’s winning states and then the two restricted games are composed with each other. At this point one can solve the resulting game through various means which are discussed in Section 4.3.2.

The final section of this chapter talks about optimizing the parametric representation of a transition function obtained from some safety property or obligation property (which is an implication between two safety properties).

### 5.12 Optimizing the Parametric Representation

The parametric representation of a transition function discussed in the previous sections is such that only the previous values of the signals in \(X'\) are remembered (the ones that are really needed) and no values from further back need to be remembered. However, in many cases remembering the individual values may not be optimal, rather remembering various Boolean functions of variables in \(X'\) yields a more economical parametric representation. This section discusses a retiming heuristic to detect such functions and obtain a better parametric representation.

**Retiming** is the technique of moving the structural location of latches (registers) in a digital circuit to improve its performance, area, and/or power characteristics in such a way that preserves its functional behavior at its outputs. (The idea was first proposed in [LS91]). In the context of synthesis, the objective is to use retiming to reduce the number of latches in the parametric representation. The following example describes a trivial parametric representation that can be optimized by retiming.
Example 5.12.1. Given the set of signals $X_{ed} = \{a, b\}$, $X_s = \{x\}$, $X_{ep} = \{c\}$, $X'_{ed} = \{a', b'\}$, and $X'_{s} = \{x'\}$, let $x' = (a \land c) \lor b'$ be the transition function that guarantees the satisfaction of some $\omega$-regular safety property. The implementation of the function $x'$ requires one to remember two individual values of $a'$ and $c'$. However, one does not need the individual values $a$ and $c$ rather the function $a' \land c'$ needs to be remembered. Therefore, $x' = f \lor b'$, where $f' = a' \land b'$ and $f$ denotes the previous value of $f'$.

When optimizing the parametric representation of a transition function, it is not always possible to assign values to the parameters early. However, when it is possible then one can achieve better results. The following example illustrates when a parameter may be assigned early.

Example 5.12.2. Let the set of signals $X_{ed} = \{a, b\}$, $X_s = \{x\}$, $X_{ep} = \{c\}$, $X'_{ed} = \{a', b'\}$, $X'_{s} = \{x'\}$, $X_{ep} = \{c'\}$, and a set of parameters $P = \{x_i\}$ and let $x' = (a \land c \land x_i) \lor b'$ be a parametric representation that captures all the system’s winning strategies for some $\omega$-regular safety property. Suppose the winning strategy for some progress property is $x_i = a \land b \land \neg b'$. This strategy is possible within the non-optimized parametric representation. The retiming should yield an optimized representation where this strategy is still possible. In this particular case, $x_i$ cannot be assigned early as it will prevent the system from making its choice after knowing the value of $b'$.

However, when the variables are partitioned such that $X_{ed} = \emptyset$ and $X_{ep} = \{a, b, c\}$, the system can decide the value of the letters in $i^{th}$ turn by setting the value of $x_i$ in the $(i-1)^{th}$ turn without compromising its ability to find the strategies for the progress properties. (This is because when playing the progress properties the $X_{ed}$ remains empty.) In this case, the new representation is $x' = f \lor b'$, where $f' = a' \land c' \land x_i$ and $f$ is the previous value of $f'$. In such a case, the system would not have a winning strategy for the progress property in either non-optimized or optimized parametric representation. The main objective of the retiming step is to obtain an optimized parametric representation which retains all the strategies through which the system can satisfy the progress properties of the specification.

Retiming may not always be successful; whenever the number of Boolean functions to be remembered is smaller than the number of individual signals that need to be remembered then an optimized parametric representation can be obtained. The following example is an illustration of this observation.
Example 5.12.3. Given the set of signals \( X_{ed} = \{a, b\} \), \( X_s = \{x\} \), \( X_{ep} = \{c\} \), \( X'_{ed} = \{a', b'\} \), \( X'_s = \{x'\} \), \( X_{ep} = \{c'\} \), and a set of parameters \( P = \{x_i\} \). Let

\[
x' = (a \land b \land c) \lor (~a \land ~b \land c) \lor (~a \land b \land ~c) \lor (a \land ~b \land ~c \land x_i) \lor b'
\]

be a parametric representation that captures all the system's winning strategies for some \( \omega \)-regular safety property. Retiming of this parametric representation will yield the original representation because 4 different Boolean functions of the three signals \( \{a, b, c\} \) need to be remembered. In this case remembering the individual values yields the best representation.

The general concept is that a combinational network is created from the outputs in \( X'_s \) where the variables in \( \hat{X} \cup X'_{ed} \) are treated as inputs (as it has been established that the inputs \( X'_{ep} \) do not appear in the support of the outputs in \( X'_s \)). As it has already been discussed that in the parametric transducer \( M_\psi \) the variables \( \hat{X} \) are outputs of latches (the inputs of these latches are the variables \( \hat{X}' \)) the objective is to move the latches towards the outputs \( X'_s \) in order to reduce the number of latches required by the parametric representation (for instance it has been shown in Example 5.12.3 that instead of latching the inputs \( a' \) and \( b' \) one could latch the function \( a' \land b' \)). In the remainder of this section the retiming heuristic is discussed along with the various challenges that must be overcome in reducing the number of latches in the parametric transducer.

To understand the retiming heuristic presented here, one must first understand the basic concept of decomposing a Boolean formula and then obtaining a directed acyclic graph (DAG) from it. A Boolean function \( f(x_1, x_2, \ldots x_n) \) (when the context is clear the Boolean function \( f(x_1, x_2, \ldots x_n) \) is written as \( f \)) can be decomposed such that \( f \) is a Boolean combination of smaller Boolean formulae \( \{f_1, f_2, \ldots f_n\} \). The decomposition is then applied to each function \( f_i \in \{f_1, f_2, \ldots, f_n\} \). The decomposition stops when the formula represented by the child node is a simple formula (for instance the formula is a single variable \( v \) or its negation).

The DAG \( D \) for a formula \( f \) has two type of nodes formula nodes and variable nodes. The formula nodes are labeled with some Boolean formula and the variable nodes are labeled with some Boolean variable. A formula node cannot be a leaf node of the DAG and every variable node in the DAG can only be a leaf
node in the DAG. A formula node \( n_f \) can be parent of a variable node \( n_v \) iff the variable \( v \) is in support of formula \( f \). A DAG is created from the Boolean decomposition of \( f \) where the root node is labeled with the formula \( f \) and its \( n \) children are each uniquely labeled with a formula in \( \{f_1, f_2, \ldots, f_n\} \). Similarly, the node labeled with the formula \( f_i \) has its children uniquely labeled with the formulae from the decomposition of \( f_i \). A new child node is not created if a node with the same label already exists. (Therefore, a child node may have more than one parent.) When the decomposition stops with respect to some predefined criteria (for instance the formulae labeling the nodes at the bottom of the DAG are smaller than some predefined size limit) variable nodes are created (if these do not exist already) and childless formula nodes are connected with the variable nodes such that the variable appears in its support. One can now create a flow network where root node is connected to a source node and the all leaf nodes are connected to a sink node.

Similarly, one can create a flow network from the solution of variables in \( X'_s \). (These are solutions computed in sections 5.7 and 5.8). Let \( \{f_{x_1}, f_{x_2}, \ldots, f_{x_n}\} \) be the solution of variables in \( X'_s \) then a DAG \( D_{x_1} \) is created for \( f_{x_1} \). The formula nodes of DAG \( D_{x_1} \) are then used as a guide to decompose \( f_{x_2} \) so that least amount of new nodes (with new labels) are created. The DAG \( D_{x_1} \) and \( D_{x_2} \) are then combined. The process is repeated until all the formulae in \( \{f_{x_1}, f_{x_2}, \ldots, f_{x_n}\} \) have been decomposed and a DAG with \( n \) (or fewer) roots is created. The roots are then connected to a source and all the leaf nodes are then connected to a sink.

After the creation of the DAG, a new DAG is obtained from it. The following rules describe how a new DAG is obtained.

1. Every formula node of the tree is traversed and if the formula represented by it has some variable \( v \in C \) (where \( C = X'_{ed} \cup P \)) in its support then the node is disconnected from its parent. (Every node which does not have a parent is called an orphanned node.)

2. Every non-orphaned child of an orphaned node is made the child of the source node. (If the child node had some variable \( v \in C \) in its support then the node would already be an orphaned node.)

3. The modified DAG can then be simplified by removing orphaned nodes.

When playing progress properties, the system chooses the value of parameters \( P \) after the environment
discloses its choice of $X'_{ed}$ variables. There exist specifications in which the system does not require the knowledge of the values assigned to variables in $X'_{ed}$ to make the correct selection of values for parameters $P$. In such cases $C = X'_{ed}$. However, detection of such cases requires extra work and therefore the current implementation of this heuristic lets the user inform the implementation what $C$ should be (by default $C = X'_{ed} \cup P$).

The Ford-Fulkerson’s maxflow algorithm [FF56] is used to find the mincut of the DAG. (Edges closer to the source are preferred over the edges closer to the sink when computing the mincut.) The label of the target node of every edge included in the mincut represents the Boolean function that needs to remembered. The following example describes a DAG created from a Boolean function.

**Example 5.12.4.** Consider the Example 5.12.1 where $x' = (a \land c) \lor b'$. The DAG shown in Figure 5.15(a) is derived from the symbolic representation of this function. The modified DAG is shown in Figure 5.15(b).

The mincut of the modified DAG is a single edge between the source node and the node labeled $a \land c$. Therefore, the formula of this node needs to be remembered.

The solution of coloring variables $X'_{K}$ is not part of the flow network as the solution of these variables is such that only variables in $X'_{ed} \cup X'_{s} \cup X'_{ep}$ appear in its support in most cases. (In Section 5.7, it was mentioned that the solution of variables in $X'_{K}$ may contain the variables $X'_{s}$. The solution of variables
in $X'_K$ are updated by replacing the variables in $X'_s$ with their solution. If retiming is performed then the solutions of variables in $X'_K$ are updated by replacing the variables $X'_s$ with the retimed solutions.)

The next section describes the challenge in creating the flow network from the BDD representation of a function.

### 5.12.1 Flow Networks and Binary Decision Diagrams

The BDD of a function is not suitable for creating the flow network. It is in common knowledge that creating a circuit for an output directly from its BDD representation often yields an implementation that requires significantly more combinational logic (this observation is also reported in [EKH12]). The following example shows that the flow network derived from a BDD representation may not be able to detect the least number of functions that need to be remembered.

**Example 5.12.5.** Consider the function $x'$ from the Example 5.12.1. The BDD for the variable order $b', a, c$ is shown in Figure 5.16(a). The DAG created from this BDD is shown in Figure 5.15(a).

The BDD for the variable order $a, b', c$ is shown in Figure 5.16(b). The DAG created from this BDD is shown in Figure 5.16(c). The modified DAG derived from Figure 5.16(c) is shown in Figure 5.16(d). One can easily verify that the mincut of this modified DAG contains two edges. Therefore, according to this mincut set the values of $a$ and $c$ need to be remembered. However, it has already been shown that one only needs to remember the function $a \land b$.

Finally one observes that the retiming heuristic is dependant on the combinational decomposition of the output functions. One may not be able to find the best way to optimize the parametric representation if the output functions have not been decomposed correctly. It is not yet known how to decompose the output functions that will enable one to find the best way to optimize the parametric representation.

### 5.12.2 Factorization and Flow Network

In the previous section it was observed that creating a flow network from the BDD of a function may not yield optimal results. In this section, a heuristic for creating a more suitable flow network is proposed.
This heuristic is based on the factorization of a function [Min93]. Given a Boolean function, one finds a divisor such that only the past variables in X appear in its support. The function \( f \) is rewritten as \( (D \land Q) \lor R \).

The node representing \( f \) has three children \( Q, D, \) and \( R \). Each child node then gets factored. Once the DAG is created, it gets modified through the same rules described earlier with one extra rule: if a node is orphaned and it is still connected to the \( D \)-child and \( Q \)-child then a node is created with the formula \( D \land Q \) as its label, this node is then made the child of the source node.

The following example illustrates the flow network of a function created by factorizing the function through the above described procedure.

**Example 5.12.6.** Given the set of signals \( X_{ed} = \{a, b\}, X_s = \{x\}, X_{ep} = \{c\}, X'_{ed} = \{a', b'\}, \) and \( X'_{s} = \{x'\} \). Let \( x' = ((a \lor c) \land b') \lor ((\neg a \lor b) \land \neg b') \) be the solution of some \( \omega \)-regular safety property. A DAG based on factorization of \( x' \) is shown on the left in Figure 5.17. The modified DAG is shown on the right in Figure 5.17. One can verify that the mincut of the modified DAG is 2 and one only needs to remember \( a \lor c \) and \( \neg a \lor b \).

When the set \( X'_s \) is not a singleton (there are multiple outputs) then the factorization procedure proceeds greedily. A function is selected randomly then the divisors used to factorize this function are stored to be used for factorizing other functions. Another function is then selected randomly, it is then factorized.
using the already stored divisors. If these divisors are not adequate then new divisors are created and stored. The efficiency of the factorization-based approach depends on the quality of the divisors and the selection order of the functions. The current implementation of this heuristic attempts to compute a divisor $D$ such that the decomposition of the formula $F = (D \land Q) \lor R$ yields an $R$ which is a small formula.

5.12.3 Retiming and Automata

One may wonder whether the retiming heuristic may be useful when synthesizing directly from automata that accept the safety properties. The retiming heuristic presented here is not very effective without the framework discussed in this chapter. The following example illustrates the reason for its ineffectiveness.

Example 5.12.7. Given the set of signals $X_{ed} = \{h\}$, $X_s = \{x, y\}$, $X_{ep} = \emptyset$ and $P = \{x_i, y_i\}$. Consider the specification with two properties $G(h \rightarrow X x)$ and $G(\neg h \rightarrow X y)$.

The reduced automaton for the first property has two states encoded by the variable $s_1$, where $s_1$ is asserted when $h$ is asserted in the previous turn. The parametric solution of $x = s_1 \lor x_i$. Similarly, the reduced automaton for the second property has two states encoded by the variable $s_2$ where $s_2$ is asserted.
when \( h \) is de-asserted in the previous turn. The parametric solution for \( y = s_2 \lor y_i \).

As one can observe that the system only needs to remember the previous value of \( h \), however, the two automata do not realize that both are remembering similar information; that both variables \( s_1 \) and \( s_2 \) are mutually exclusive. The retiming heuristic discussed in this section is not equipped to take into consideration the reachability information and thus is not capable of detecting such cases.

However, the relation \( R \) obtained from these two properties is such that the synthesis procedure is able to detect the fact that only the previous value of \( h \) needs to be remembered and one can synthesize a parametric representation which only uses a single latch.

\[
x = h_L \lor x_i
\]
\[
y = \neg h_L \lor y_i
\]

where \( h_L \) represents the latch that remembers the previous value of \( h \).

One can rectify the problems discussed Example 5.12.7 by re-encoding the reachable state space. However, this is the exact problem that has been discussed in the Section 5.1; re-encoding the state space often leads to other inefficiencies. The following example illustrates the fact that when converting safety properties to automata, retiming has already occurred during conversion for each property.

**Example 5.12.8.** Consider the property

\[
\varphi = \varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_4
\]
\[
\varphi_1 = G((a \land c) \rightarrow X x)
\]
\[
\varphi_2 = G((\neg a \land b) \rightarrow X \neg x)
\]
\[
\varphi_3 = G((b \land \neg c) \rightarrow X y)
\]
\[
\varphi_4 = G((\neg b \land a) \rightarrow X \neg y),
\]

where \( X_{ed} = \{a, b, c\}, X_s = \{x, y\}, X_{ep} = \emptyset \), and the set of parameters \( P = \{x_i, y_i\} \). (One can verify that this specification is realizable.) The automaton for each property contains only one state variable, which remembers if the condition in the antecedent held (i.e., in the first property the state variable remembers if
\( a \land c \) occurred in the previous turn). The parametric representation for each property obtained from the respective automaton is as follows:

\[
x' = s_1 \lor x_i \quad \text{where} \quad s_1 = a \land c
\]

\[
x' = \neg s_2 \land x_i \quad \text{where} \quad s_2 = \neg a \land b
\]

\[
y' = s_3 \lor y_i \quad \text{where} \quad s_3 = b \land c
\]

\[
y' = \neg s_4 \land y_i \quad \text{where} \quad s_4 = \neg b \land a .
\]

When these four parameterized representations are conjoined, the resulting representation contains four state variables. At this point one needs a way to reverse the effects of retiming (that was done during the conversion of the property to an automaton) to discover that this parametric representation can be optimized by using only three state variables which remember the previous value of the three inputs \( \{a, b, c\} \).

On the other hand, the parametric representation for each property obtained from the relation-based approach is as follows:

\[
x' = (a \land c) \lor x_i
\]

\[
x' = (a \lor \neg b) \land x_i
\]

\[
y' = (b \land c) \lor y_i
\]

\[
y' = (b \lor \neg a) \land y_i .
\]

When the parameterized representations of these four properties are conjoined, the resulting representation contains only three state variables. This is because in the resulting parametric representation there is direct way to reuse the state variables. Finally, when the retiming heuristic is applied to this parametrized representation, one comes to the conclusion that one cannot use fewer latches than three to implement this parametric representation.

In model-checking the model is verified against each property individually through its automaton (the efficiency of the verification step demands the best encoding of the automaton). On the other hand, all the automatic synthesis approaches require all the properties of the specification to be considered together (at
least in the final step). The relation-based approach has been developed with this difference in mind, that is why the retiming step is delayed. Therefore, the retiming is done once all the safety properties have been dealt with. (Retiming can also be done after the strategies for progress properties have been found.) This delay in retiming is one of the reasons why the relation-based approach scales so well.

This chapter has described a new approach to synthesize safety properties in the specification. The new approach is instrumental in reducing the size of the overall game and then having significant impact on the quality of results once the specification is proven to be realizable. The key strength of this approach is its ability to obtain a small safety game when the number of safety properties in the specification is quite large.
In the existing literature, the focus has been on turn-based games. However, the specification of a reactive-system is naturally translated into an input-based game. Given an input-based game the symbolic representation of the corresponding turn-based game is often inefficient. The inefficiencies cause the game solving algorithms to work harder in solving a turn-based game compared to its input-based counterpart. In Section 6.1, the correspondence between the two games has been established. The attraction based turn-based game solving algorithms cannot be used to solve the input-based games directly. In Section 6.2, the changes necessary to make this algorithms solve input-based games have been discussed.

6.1 Input-Based Games vs. Turn-Based Games

This section describes the correspondence between the input-based games and the turn-based games. Given an input-based game, one can always obtain a turn-based game from it. There are various ways in which a turn-based game can be obtained from an input-based game; one of these has been discussed here.

Definition 6.1.1. Turn-based game $\hat{G} = \{Q_0, Q_1, \delta, \alpha\}$ is the associate of input-based game $G = \{\Sigma, Q, D, \delta, \alpha\}$ if the following conditions hold. $\hat{Q} = \hat{Q}_0 \cup \hat{Q}_1$ with

$$\hat{Q}_0 = Q \cup \{(q, \sigma_{ed}, \sigma_s) \mid \exists \sigma_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D\},$$

$$\hat{Q}_1 = \{(q, \sigma_{ep}, \sigma_s, \sigma_{ep}) \mid (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D\} \cup$$

$$\{(q, \sigma_{ed}) \mid \exists \sigma_s \cdot \exists \sigma_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D\}.$$
The states in $Q \subseteq \hat{Q}$ are the base states of $\hat{G}$. The transition function $\hat{\delta} : \hat{Q} \rightarrow 2^{\hat{Q}}$ is defined by:

$$\hat{\delta}(q) = \{(q, \sigma_{ed}) | \exists \sigma_{s}. \exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_{s}, \sigma_{ep})) \in D\},$$

$$\hat{\delta}((q, \sigma_{ed})) = \{(q, \sigma_{ed}, \sigma_{s}) | \exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_{s}, \sigma_{ep})) \in D\},$$

$$\hat{\delta}((q, \sigma_{ed}, \sigma_{s})) = \{(q, \sigma_{ed}, \sigma_{s}, \sigma_{ep}) | (q, (\sigma_{ed}, \sigma_{s}, \sigma_{ep})) \in D\},$$

$$\hat{\delta}((q, \sigma_{ed}, \sigma_{s}, \sigma_{ep})) = \{\hat{\delta}(q, (\sigma_{ed}, \sigma_{s}, \sigma_{ep}))\}.$$

Let $\gamma : \hat{Q} \rightarrow Q$ be the projection function that maps each base state to itself and any other state of turn-based game $\hat{G}$ to its first component. One can write $\gamma(S)$ for the point-wise extension of $\gamma$ to sets of states. Let $\Gamma : \hat{Q}^{\omega} \rightarrow Q^{\omega}$ map a sequence of states of $\hat{G}$ to the result of applying $\gamma$ to every fourth state:

$$\Gamma(\hat{\rho})_n = \gamma(\hat{\rho}_{4n}).$$

Then,

$$\hat{\alpha} = \{\hat{\rho} \in \hat{Q}^{\omega} | \Gamma(\hat{\rho}) \in \alpha\}$$

is the winning condition of turn-based game $\hat{G}$. If $\hat{G}$ is the associate of $G$, then it is also said that $G$ is the associate of $\hat{G}$.

The associate game $\hat{G}$ is four-partite. If $G$ is a game with generalized parity winning condition $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$ then $\hat{\alpha}$ can be written as $\hat{\Pi} = \{\hat{\pi}_1, \hat{\pi}_2, \ldots, \hat{\pi}_k\}$, where

$$\forall q \in \hat{Q}. \hat{\pi}_k(q) = \pi_k(\gamma(q)).$$

Later in this section (Theorem 6.1.7) it shall be seen that associate $\hat{G}$ can be solved instead of the input-based game $G$. Notice that the input alphabet $\Sigma$ is dropped from the definition of an associate game. One can verify that the input-alphabet is redundant in the definition of an associate game. The transition function of the associate game is deterministic iff the transition function of the input-based game is deterministic. Furthermore, the initial state is not mentioned in the definitions of either the input-based game or the associate game. The initial state has been dropped from these definitions to simplify the proofs. One can verify that there is a unique initial state in either game and the initial state belongs to Player 0.
The input-based game and its associate turn-based game can be played interchangeably. It is now proven that the strategy for either player in one game can be converted for the associate game without affecting the outcome of the game.

**Definition 6.1.2.** Let $G = (\Sigma, Q, D, \delta, \alpha)$ be an input-based game and $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\alpha})$ its associate game. Given Player 1’s strategy $\hat{\tau}_1 : S_1 \times \hat{Q}_1 \rightarrow S_1 \times \hat{Q}_0$ for $\hat{G}$, then

$$\tau_1 : S_1 \times Q \times \Sigma_{ed} \rightarrow S_1 \times \Sigma_s$$

is Player 1’s *associate* strategy of $\hat{\tau}_1$ defined as follows. Let

$$\hat{\tau}_1(s_1^1, (q, \sigma_{ed})) = (s_1^2, (q, \sigma_{ed}, \sigma_s)) \ ,$$

$$\hat{\tau}_1(s_1^2, (q, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_1^3, q') \ ,$$

then

$$\tau_1(s_1^1, q, \sigma_{ed}) = (s_1^3, \sigma_s) \ .$$

Given Player 0’s strategy $\hat{\tau}_0 : S_0 \times \hat{Q}_0 \rightarrow S_0 \times \hat{Q}_1$ for $\hat{G}$, then

$$\tau_0^1 : S_0 \times Q \rightarrow \Sigma_{ed}$$

$$\tau_0^2 : S_0 \times Q \times \Sigma_{ed} \times \Sigma_s \rightarrow S_0 \times \Sigma_{ep}$$

is Player 0’s *associate* strategy of $\hat{\tau}_0$ defined as follows. Let

$$\hat{\tau}_0(s_0^1, q) = (s_0^2, (q, \sigma_{ed})) \ \text{and} \$$

$$\hat{\tau}_0(s_0^2, (q, \sigma_{ed}, \sigma_s)) = (s_0^3, (q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \ ,$$

then

$$\tau_0^1(s_0^1, q) = \sigma_{ed} \ \text{and} \$$

$$\tau_0^2(s_0^1, q, \sigma_{ed}, \sigma_s) = (s_0^3, \sigma_{ep}) \ .$$

Given Player 1’s strategy $\tau_1 : S_1 \times Q \times \Sigma_{ed} \rightarrow S_1 \times \Sigma_s$ for $G$,

$$\hat{\tau}_1 : S_1 \times \hat{Q}_1 \rightarrow S_1 \times \hat{Q}_0$$
is Player 1’s associate strategy of $\tau_1$ for $G$ defined as follows. Let

$$\tau_1(s_1^1, q, \sigma_{ed}) = (s_1^2, \sigma_s) ,$$

then

$$\hat{\tau}_1(s_1^1, (q, \sigma_{ed})) = (s_1^2, (q, \sigma_{ed}, \sigma_s)) ,$$

while the strategy for a state $q \in Q \times \Sigma_{ed} \times \Sigma_s \times \Sigma_{ep}$ is determined by the transition function $\delta$ as follows:

$$\hat{\tau}_1(s_1^1, (q, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_1^2, \delta(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))) .$$

Given Player 0’s strategy $\tau_0^1 : S_0 \times Q \to \Sigma_{ed}$ and $\tau_0^2 : S_0 \times \Sigma_{ed} \times \Sigma_s \to S_0 \times \Sigma_{ep}$ for $G$,

$$\hat{\tau}_0 : S_0 \times \hat{Q}_0 \to S_0 \times \hat{Q}_1$$

is Player 0’s associate strategy of $(\tau_0^1, \tau_0^2)$ for $G$ defined as follows. Let

$$\tau_0^1(s_0^1, q) = \sigma_{ed} ,$$

$$\tau_0^2(s_0^1, q, \sigma_{ed}, \sigma_s) = (s_0^2, \sigma_{ep}) ,$$

then

$$\hat{\tau}_0(s_0^1, q) = (s_0^1, (q, \sigma_{ed})) \quad \text{and} \quad \hat{\tau}_0(s_0^1, (q, \sigma_{ed}, \sigma_s)) = (s_0^2, (q, \sigma_{ed}, \sigma_s, \sigma_{ep})) .$$

If $\tau_1$ is Player i’s strategy for $G$, then $AS(\tau_1)$ is Player i’s associate strategy for the associate game $\hat{G}$. If $\hat{\tau}_1$ is Player i’s strategy for $\hat{G}$ then $AS(\hat{\tau}_1)$ is Player i’s associate strategy for the associate game $\hat{G}$. Note that a strategy and its associate use the same memory. In the rest of the section, if $\rho$ is a play of $\hat{G}$ then $\rho_{l,m}$ is a segment of $\rho$ such that $\rho_{l,m} = \rho_l, \rho_{l+1}, \ldots, \rho_{m-1}, \rho_m$.

It is now shown that the strategy $AS(\tau_1)$ is winning for Player i iff $\tau_1$ is. The following two lemmas prove that using associate strategies preserves the outcome of a play.

**Lemma 6.1.3.** Let $G = (\Sigma, Q, D, \delta, \alpha)$ be an input-based game and $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\alpha})$ be its associate game. Let $\hat{\rho}$ be the play of $\hat{G}$ starting from base state $q \in Q$ when, for $i \in \{0, 1\}$, Player $i$ plays according
to strategy $\hat{\tau}_i$. If $\rho$ is the play of $G$ starting from $q$ when Player $i$ plays according to strategy $\Delta S(\hat{\tau}_i)$, then $\rho = \Gamma(\hat{\rho})$.

**Proof.** For the play $\rho$ of $G$, let $\rho^S_i$ be the corresponding sequence of Player $i$’s memory states. For the play $\hat{\rho}$ of $\hat{G}$, let $\hat{\rho}^S_i$ be the corresponding sequence of Player $i$’s memory states. It is now proven by induction on length of the play $\rho$ and the corresponding sequence of Player $i$’s memory states $\rho^S_i$ that $\rho = \Gamma(\hat{\rho})$ and $\forall n \in \mathbb{N}. \rho^S_i_n = \hat{\rho}^S_i_n$. Since the initial states of the runs $\hat{\rho}$ and $\rho$ are the same, while Player $i$’s memory is always initialized to $\tilde{s}_i$, the base case holds.

For the inductive step, let $\rho_{0,n} = \Gamma(\hat{\rho}_{0,4n})$ and $\forall x.(0 \leq x \leq n) \rightarrow \rho^S_i_x = \hat{\rho}^S_i_x$. Let

\[
\hat{\rho}_{4n} = \rho_n = u \\
\hat{\rho}_{4n}^S_0 = \rho^S_0 = s^1_0 \quad \text{and} \\
\hat{\rho}_{4n}^S_1 = \rho^S_1 = s^1_1 .
\]

Suppose Player $i$ plays according to

\[
\hat{\tau}_0(s^1_0, u) = (s^2_0, (u, \sigma_{ed})) , \\
\hat{\tau}_1(s^1_1, (u, \sigma_{ed})) = (s^2_1, (u, \sigma_{ed}, \sigma_s)) , \\
\hat{\tau}_0(s^2_0, (u, \sigma_{ed}, \sigma_s)) = (s^3_0, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) \quad \text{and} \\
\hat{\tau}_1(s^2_1, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s^3_1, v) .
\]

then

\[
\hat{\rho}_{4n+1} = (u, \sigma_{ed}) , \quad \hat{\rho}^S_0_{4n+1} = s^2_0 , \hat{\rho}^S_1_{4n+1} = s^1_1 , \\
\hat{\rho}_{4n+2} = (u, \sigma_{ed}, \sigma_s) , \quad \hat{\rho}^S_0_{4n+2} = s^2_0 , \hat{\rho}^S_1_{4n+2} = s^2_1 , \\
\hat{\rho}_{4n+3} = (u, \sigma_{ed}, \sigma_s, \sigma_{ep}) , \quad \hat{\rho}^S_0_{4n+3} = s^3_0 , \hat{\rho}^S_1_{4n+3} = s^2_1 \\
\]

and

\[
\hat{\rho}_{4n+4} = v , \quad \hat{\rho}^S_0_{4n+4} = s^3_0 , \hat{\rho}^S_1_{4n+4} = s^3_1 .
\]
Let $\tau_i = \text{AS}(\hat{\tau}_i)$. By Definition 6.1.2

$$
\tau_0^1(s_0^1, u) = \sigma_{ed},
$$
$$
\tau_1(s_1^1, u, \sigma_{ed}) = (s_1^3, \sigma_s) \quad \text{and}
$$
$$
\tau_0^2(s_0^1, u, \sigma_{ed}, \sigma_s) = (s_0^3, \sigma_{ep}).
$$

By Definition 6.1.1, $\delta(u, (\sigma_{ed}, \sigma_s, \sigma_{ep})) = v$. Hence $\rho_{n+1} = v$, $\rho_n^{s_0} = s_0^3$ and $\rho_n^{s_1} = s_1^3$. It follows that $\rho_{0,n+1} = \Gamma(\hat{\rho}_{0,4n+4})$ and $\forall x(0 \leq x \leq n + 1) \rightarrow \rho_x^{s_i} = \rho_{4x}^{s_i}$. Therefore, it is concluded that $\rho = \Gamma(\hat{\rho})$ which by definition of $\hat{\alpha}$ implies that $\hat{\rho}$ is winning iff $\rho$ is winning.

\[ \square \]

**Lemma 6.1.4.** Let $G = (\Sigma, Q, D, \delta, \alpha)$ be an input-based game and $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\alpha})$ be its associate game. Let $\rho$ be the play of $G$ starting from $q \in Q$ when, for $i \in \{0, 1\}$, Player $i$ plays according to strategy $\tau_i$. If $\hat{\rho}$ is the play of $\hat{G}$ starting in $q$ when Player $i$ plays according to strategy $\text{AS}(\hat{\tau}_i)$, then $\Gamma(\hat{\rho}) = \rho$.

**Proof.** For the play $\rho$ of $G$, let $\rho_n^{s_i}$ be the corresponding sequence of Player $i$’s memory states. For the play $\hat{\rho}$ of $\hat{G}$, let $\hat{\rho}_n^{s_i}$ be the corresponding sequence of Player $i$’s memory states. It is now proven on length of the play $\hat{\rho}$ and the corresponding sequence of Player $i$’s memory states $\hat{\rho}_n^{s_i}$ that $\rho = \Gamma(\hat{\rho})$ and $\forall n \in N \rightarrow \rho_n^{s_i} = \hat{\rho}_n^{s_i}$. Since the starting states of both runs $\rho$ and $\hat{\rho}$ are the same, while the memory is always initialized to $\tilde{s}_i$, the base case holds. For the inductive step, let $\Gamma(\hat{\rho}_{0,4n}) = \rho_{0,n}$ and $\forall x.(0 \leq x \leq n) \rightarrow \rho_{4x}^{s_i} = \rho_x^{s_i}$. Let

$$
\rho_n = \hat{\rho}_{4n} = u
$$
$$
\rho_n^{s_0} = \hat{\rho}_{4n}^{s_0} = s_0^1 \quad \text{and}
$$
$$
\rho_n^{s_1} = \hat{\rho}_{4n}^{s_1} = s_1^1.
$$

Suppose

$$
\tau_0^1(s_0^1, u) = \sigma_{ed},
$$
$$
\tau_1(s_1^1, u, \sigma_{ed}) = (s_1^3, \sigma_s),
$$
$$
\tau_0^2(s_0^1, u, \sigma_{ed}, \sigma_s) = (s_0^3, \sigma_{ep}) \quad \text{and}
$$
$$
\delta(u, (\sigma_{ed}, \sigma_s, \sigma_{ep})) = v,
$$

then $\rho_{0,n+1} = \Gamma(\hat{\rho}_{0,4n+4})$ and $\forall x(0 \leq x \leq n + 1) \rightarrow \rho_x^{s_i} = \rho_{4x}^{s_i}$. Therefore, it is concluded that $\rho = \Gamma(\hat{\rho})$ which by definition of $\hat{\alpha}$ implies that $\hat{\rho}$ is winning iff $\rho$ is winning.
then $\rho_{n+1} = v$, $\rho S_0 = s_0^2$ and $\rho S_1 = s_1^2$. Let $\hat{\tau}_i = AS(\tau_i)$. Then, by Definition 6.1.2

$$\hat{\tau}_0(s_0^1, u) = (s_0^1, (u, \sigma_{ed})),$$

which means that

$$\hat{\rho}_{4n+1} = (u, \sigma_{ed}), \quad \hat{\rho}_{4n+1}^{S_0} = s_0^1 \quad \text{and} \quad \hat{\rho}_{4n+1}^{S_1} = s_1^1.$$

Then, by Definition 6.1.2

$$\hat{\tau}_1(s_1^1, (u, \sigma_{ed})) = (s_1^1, (u, \sigma_{ed}, \sigma_s)),$$

which means that

$$\hat{\rho}_{4n+2} = (u, \sigma_{ed}, \sigma_s), \quad \hat{\rho}_{4n+2}^{S_0} = s_0^1 \quad \text{and} \quad \hat{\rho}_{4n+2}^{S_1} = s_1^1.$$

Then, by Definition 6.1.2

$$\hat{\tau}_0(s_0^2, (u, \sigma_{ed}, \sigma_s)) = (s_0^2, (v, \sigma_{ed}, \sigma_s, \sigma_{ep})),$$

which means that

$$\hat{\rho}_{4n+3} = (u, \sigma_{ed}, \sigma_s, \sigma_{ep}), \quad \hat{\rho}_{4n+3}^{S_0} = s_0^2 \quad \text{and} \quad \hat{\rho}_{4n+3}^{S_1} = s_1^1.$$

Finally, by Definition 6.1.2

$$\hat{\tau}_0(s_1^2, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_1^2, v),$$

which means that

$$\hat{\rho}_{4n+4} = v, \quad \hat{\rho}_{4n+4}^{S_0} = s_0^2 \quad \text{and} \quad \hat{\rho}_{4n+4}^{S_1} = s_1^2.$$

It follows that $\Gamma(\hat{\rho}_{4n+4}) = \rho_{0,n+1}$ and $\forall x \ (0 \leq x \leq n + 1) \rightarrow \hat{\rho}_{4x}^{S_i} = \rho_x^{S_i}$. Therefore, it is concluded that $\Gamma(\hat{\rho}) = \rho$, which by definition of $\hat{\alpha}$ implies that $\hat{\rho}$ is winning iff $\rho$ is winning.

If $\tau_i$ is a strategy for Player $i$ in an input-based game $G$, then Lemma 6.1.5 proves that $AS(AS(\tau_i)) = \tau_i$. On the other hand if $\hat{\tau}_i$ is a strategy for Player $i$ in the associate game $\hat{G}$, then Lemma 6.1.6 proves that even though $AS(AS(\hat{\tau}_i))$ may differ from $\hat{\tau}_i$, the two strategies are interchangeable.

**Lemma 6.1.5.** Let $G = (\Sigma, Q, D, \delta, \alpha)$ be an input-based game and $\hat{G} = (Q, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\alpha})$ be its associate game. If $\tau_i$ is the strategy of Player $i$ in $G$, then $AS(AS(\tau_i)) = \tau_i$. 
Proof. Suppose

\[
\tau_0^1(s_0, u) = \sigma_{ed}, \\
\tau_1(s_1, u, \sigma_{ed}) = (s_1^2, \sigma_s) \quad \text{and} \\
\tau_0^2(s_0, u, \sigma_{ed}, \sigma_s) = (s_0^2, \sigma_{ep}).
\]

By Definition 6.1.2

\[
\text{AS}(\tau_0)(s_0^1, u) = (s_0^1, (u, \sigma_{ed})), \\
\text{AS}(\tau_1)(s_1^1, (u, \sigma_{ed})) = (s_1^1, (u, \sigma_{ed}, \sigma_s)), \\
\text{AS}(\tau_0)(s_0, u, \sigma_{ed}, \sigma_s) = (s_0^2, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) \quad \text{and}, \\
\text{AS}(\tau_1)(s_1, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_1, v).
\]

By Definition 6.1.2

\[
\text{AS}(\text{AS}(\tau_0^1))(s_0^1, u) = \sigma_{ed}, \\
\text{AS}(\text{AS}(\tau_1))(s_1^1, u, \sigma_{ed}) = (s_1^2, \sigma_s) \quad \text{and} \\
\text{AS}(\text{AS}(\tau_0^2))(s_0^1, u, \sigma_{ed}, \sigma_s) = (s_0^2, \sigma_{ep}).
\]

Hence \(\text{AS}(\tau_i) = \tau_i\). \(\square\)

Lemma 6.1.6. Let \(G = (\Sigma, Q, D, \delta, \alpha)\) be an input-based game and \(\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\alpha})\) be its associate. Let \(\hat{\rho}\) be the play of \(\hat{G}\) starting in \(q \in \hat{Q}\) when Player 0 plays according to \(\hat{\tau}_0\) and Player 1 plays according to \(\hat{\tau}_1\). For \(i \in \{0, 1\}\) let \(j = \neg i\) and let \(\rho^i\) be the play of \(\hat{G}\) starting in \(q \in \hat{Q}\) by Player \(i\) playing according to \(\hat{\tau}_i\) and Player \(j\) playing according to \(\text{AS}(\text{AS}(\hat{\tau}_j))\). Then, \(\rho^0 = \rho^1 = \hat{\rho}\).

Proof. For the play \(\hat{\rho}\) of \(\hat{G}\), let \(\hat{\rho}^{S_i}\) be the corresponding sequence of Player \(i\)'s memory states. For the play \(\rho^0\) of \(\hat{G}\), let \(\rho^{0,S_i}\) be the corresponding sequence of Player \(i\)'s memory states. Likewise, for the play \(\rho^1\) of \(\hat{G}\), let \(\rho^{1,S_i}\) be the corresponding sequence of Player \(i\)'s memory states. It is now proven by induction on the length of the runs and the corresponding sequence of Player \(i\)'s memory states that for \(p \in \{0, 1\}\), \(\hat{\rho} = \rho^p\) and \(\forall n \in N. \hat{\rho}^{S_i}_{4n} = \rho^{p,S_i}_{4n}\).
Since both runs start in the same state, while the memory is always initialized to $\tilde{s}_i$, the base case holds. For the inductive step, let $\hat{\rho}_{0,4n} = \rho^p_{0,4n}$ and $\forall x. 0 \leq x \leq n \rightarrow \hat{\rho}_{4x}^{S_i} = \rho^p_{4x}^{S_i}$. Suppose $\hat{\rho}_{4n} = u$, $\hat{\rho}_{4n}^{S_0} = s_0^1$ and $\hat{\rho}_{4n}^{S_1} = s_1^1$; then $\hat{\rho}_{4n} = u$, $\rho_{4n}^{p,S_0} = s_0^1$ and $\rho_{4n}^{p,S_1} = s_1^1$. Suppose

$$\hat{\tau}_0(s_0^1, u) = (s_0^2, (u, \sigma_{ed})),$$

$$\hat{\tau}_1(s_1^1, (u, \sigma_{ed})) = (s_1^2, (u, \sigma_{ed}, \sigma_s)),$$

$$\hat{\tau}_0(s_0^2, (u, \sigma_{ed}, \sigma_s)) = (s_0^3, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) \quad \text{and,}$$

$$\hat{\tau}_1(s_1^2, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_1^3, v).$$

This implies that

$$\hat{\rho}_{4n+1} = (u, \sigma_{ed}), \quad \hat{\rho}_{4n+1}^{S_0} = s_0^1, \quad \hat{\rho}_{4n+1}^{S_1} = s_1^1,$$

$$\hat{\rho}_{4n+2} = (u, \sigma_{ed}, \sigma_s), \quad \hat{\rho}_{4n+2}^{S_0} = s_0^2, \quad \hat{\rho}_{4n+2}^{S_1} = s_1^2,$$

$$\hat{\rho}_{4n+3} = (u, \sigma_{ed}, \sigma_s, \sigma_{ep}), \quad \hat{\rho}_{4n+3}^{S_0} = s_0^3, \quad \hat{\rho}_{4n+3}^{S_1} = s_1^3,$$

and

$$\hat{\rho}_{4n+4} = v, \quad \hat{\rho}_{4n+4}^{S_0} = s_0^3, \quad \hat{\rho}_{4n+4}^{S_1} = s_1^3.$$

Definition 6.1.2

$$\text{AS}(\hat{\tau}_0^1)(s_0^1, u) = \sigma_{ed},$$

$$\text{AS}(\hat{\tau}_1)(s_1^1, (u, \sigma_{ed})) = (s_1^3, \sigma_s) \quad \text{and}$$

$$\text{AS}(\hat{\tau}_0^2)(s_0^1, (u, \sigma_{ed}, \sigma_s)) = (s_0^3, \sigma_{ep}).$$

By Definition 6.1.2

$$\text{AS}(\text{AS}(\hat{\tau}_0))(s_0^1, u) = (s_0^1, (u, \sigma_{ed})),$$

$$\text{AS}(\text{AS}(\hat{\tau}_1))(s_1^1, (u, \sigma_{ed})) = (s_1^1, (u, \sigma_{ed}, \sigma_s)),$$

$$\text{AS}(\text{AS}(\hat{\tau}_0))(s_0^1, (u, \sigma_{ed}, \sigma_s)) = (s_0^3, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})).$$
and

\[ AS(AS(\hat{\tau}_1))(s_1^1, (u, \sigma_{ed}, \sigma_s, \sigma_{ep})) = (s_3^3, v) \ . \]

If Player 1 plays according to \( AS(AS(\hat{\tau}_1)) \) and Player \( j \) plays according to \( \hat{\tau}_0 \), then

\[
\begin{align*}
\rho_{4n+1}^1 &= (u, \sigma_{ed}), & \rho_{4n+1}^{1,0} &= s_0^1, & \rho_{4n+1}^{1,1} &= s_1^1, \\
\rho_{4n+2}^1 &= (u, \sigma_{ed}, \sigma_s), & \rho_{4n+2}^{1,0} &= s_0^2, & \rho_{4n+2}^{1,1} &= s_1^2, \\
\rho_{4n+3}^1 &= (u, \sigma_{ed}, \sigma_s, \sigma_{ep}), & \rho_{4n+3}^{1,0} &= s_0^3, & \rho_{4n+3}^{1,1} &= s_1^3, \\
\rho_{4n+4}^1 &= v, & \rho_{4n+4}^{1,0} &= s_0^3, & \rho_{4n+4}^{1,1} &= s_1^3.
\end{align*}
\]

Hence \( \rho_{0,4n+4}^1 = \rho_{4n+4}^j \). Similarly, if Player 0 plays according to \( AS(AS(\hat{\tau}_0)) \) and Player 1 plays according to \( \hat{\tau}_1 \), then it follows that \( \hat{\rho}_{0,4n+4} = \rho_{0,4n+4}^0 \). Therefore, \( \rho_{0,4n+4}^j = \rho_{0,4n+4}^i \). Hence it is concluded that \( \hat{\rho} = \rho^j \).

Finally, the following theorem proves that a winning strategy for Player \( i \) in \( G \) can be translated to a winning strategy for the associate game \( \hat{G} \).

**Theorem 6.1.7.** A strategy \( \hat{\tau}_i \) for Player \( i \) in \( \hat{G} \) is winning iff the strategy \( AS(\hat{\tau}_i) \) for Player \( i \) in \( G \) is winning. Conversely, a strategy \( \tau_i \) for Player \( i \) in \( G \) is winning iff the strategy \( AS(\tau_i) \) for Player \( i \) in \( \hat{G} \) is winning.

**Proof.** For \( i \in \{0, 1\} \), let \( j = -i \), let \( \hat{\tau}_i \) be a winning strategy for Player \( i \) in \( \hat{G} \). Let Player \( i \) play according to \( AS(\hat{\tau}_i) \) in \( G \) and suppose it loses to Player \( j \)'s strategy \( \tau_j \). Let \( \hat{\rho} \) be the play of \( \hat{G} \) produced by \( AS(AS(\hat{\tau}_i)) \) and \( AS(\tau_j) \). From Lemma 6.1.4, \( \hat{\rho} \) is a losing play for Player \( i \). From Lemma 6.1.6, \( AS(AS(\hat{\tau}_i)) \) and \( \hat{\tau}_i \) produce an identical play when Player \( j \) uses a fixed strategy. Since \( \hat{\tau}_i \) is a winning strategy, a contradiction has been reached. Therefore, the assumption is wrong and there does not exist a winning strategy for Player \( j \) in \( G \) when Player \( i \) plays according to \( AS(\hat{\tau}_i) \). Hence \( AS(\hat{\tau}_i) \) is a winning strategy for Player \( i \) in \( G \).

Similarly, it is proven that \( \tau_i \) for Player \( i \) in \( G \) is winning iff the strategy \( AS(\tau_i) \) for Player \( i \) in \( \hat{G} \) is winning.
It has been shown that algorithms designed specifically for turn-based games can be adapted for input-based games. It is show in the next section that it is not necessary to explicitly play the associate turn-based game. One can implicitly play the associate turn-based game by using the representation of the input-based game.

### 6.2 Solving Input-Based Games

The turn-based game described here has significantly larger state space as size of state space is $O(|Q \times \Sigma|)$. In most conversions of input-based games to turn-based games the state space increases in proportion to the input alphabet. When deploying symbolic algorithms for playing these turn-based games, more variables are required to encode the states. It can be verified from Definition 6.1.1 that the base states that appear in a sequence are sufficient to decide if the sequence is winning. This suggests that one collapse the four transitions from a base state to another base state into one transition so as to play the associate game $\hat{G}$ without explicitly creating it. In this section, the details of this process are discussed in the context of symbolic graph algorithms.

For a symbolic implementation, the number of variables appearing in the characteristic functions is important; among other things, the search for a good BDD variable order is affected and this has a negative impact on the performance of the synthesis process. In the approach presented in this chapter, the promotion of all the input variables to state variables is avoided unlike in [PPS06, FJR09, FJR10].

Both Zielonka’s algorithm for non-generalized parity winning conditions and its extension to conjunctive parity winning conditions [CHP07] are adapted to play directly the input-based games. The **input-based Zielonka** (ib-Z) algorithm requires appropriate definition of $i$-attraction of a set and of subgame when an $i$-attraction is removed from an input-based game. The correctness of this adaptation is proven by relating the run of ib-Z on the input-based game $G$ to the run of Zielonka’s algorithm on the associate game $\hat{G}$.

The key observation is that tracking the base states of $\hat{G}$ is sufficient if the input choices of each player are properly constrained. These constraints hold in the original game and the subgames that ib-Z recursively examines. (The proper notion of a subgame of input-based games is established in Definition 6.2.7.) The main result is proven by showing that the $i$-attractions computed in $G$ and $\hat{G}$ agree on the base states and that
the subgames encountered while solving the associate game \( \hat{G} \) are related to the associates of the subgames encountered while solving \( G \).

The computation of \( i \)-attractions in input-based games relies on the pre-image operator \( \text{EX} \). The \( \text{EX} \) operator is defined so that it quantifies target states from the transition relation, but does not quantify the inputs \( \sigma \). For input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \) the pre-image of \( T \subseteq Q \) is defined by

\[
\text{EX}_G T = \{ (u, \sigma) \in D \mid \delta(u, \sigma) \in T \}.
\]

When the context is clear, \( \text{EX} \) implies \( \text{EX}_G \).

**Definition 6.2.1.** Given an input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \) and the set of states \( T \subseteq Q \), let

\[
\text{MX}_0 T = \{ q \in Q \mid \exists \sigma_{ed}, \exists \sigma_s, \exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D \}
\]

\[
\wedge (\forall \sigma_s. (\exists \sigma_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \text{EX} T))
\]

\[
\vee (\forall \sigma_{ep}. (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D)) \} ,
\]

\[
\text{MX}_1 T = \{ q \in Q \mid \forall \sigma_{ed}, \forall \sigma_s, \forall \sigma_{ep}. (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D \}
\]

\[
\vee (\exists \sigma_s. (\forall \sigma_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin (D \ \text{EX} T))
\]

\[
\wedge (\exists \sigma_{ep}. (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D)) \} .
\]

The set \( T_i \subseteq Q \) is the \( i \)-attraction of \( T \) in the input-based game \( G \) (written \( \text{attr}_i(G, T) \) as for turn-based games) if it is the least set \( Z \) such that

1. \( T \subseteq Z \) and
2. \( (Q \setminus Z) \cap \text{MX}_i Z = \emptyset \).

**Lemma 6.2.2.** Let \( \hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi}) \) be the associate game of the input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \). Let \( T \subseteq Q \) be the set of base states and let \( \hat{T} = T \). Let \( T_i \) be the states of \( G \) that can be forced by Player \( i \) into \( T \) in one or less steps and \( \hat{T}_i \) be the states of \( \hat{G} \) that can be forced by Player \( i \) into \( \hat{T} \) in four or less steps then \( T_i = Q \cap \hat{T}_i \).
Proof. Let $\hat{M}_i S$ be the set of states that Player $i$ can control to $S$ in one step. Since $\hat{G}$ is four-partite, whenever $T$ is a set of base states it follows:

\[
\hat{M}_i^0 T = T ,
\]
\[
\hat{M}_i^1 T = \{ (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q} \mid \delta((q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \in T \} ,
\]
\[
\hat{M}_i^2 T = \{ (q, \sigma_{ed}, \sigma_s) \in \hat{Q} \mid \exists \sigma_{ep} . (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{M}_i^1 T \} ,
\]
\[
\hat{M}_i^3 T = \{ (q, \sigma_{ed}) \in \hat{Q} \mid \forall \sigma_s . (q, \sigma_{ed}, \sigma_s) \not\in \hat{Q} \setminus \hat{M}_i^2 T \} ,
\]
\[
\hat{M}_i^4 T = \{ q \in Q \mid \exists \sigma_{ed} . (q, \sigma_{ed}, \sigma_s) \in \hat{M}_i^3 T \} ,
\]
\[
\hat{M}_i^1 T = \{ (q, \sigma_{ed}, \sigma_s) \in \hat{Q} \mid \forall \sigma_{ep} . (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \not\in \hat{Q} \setminus \hat{M}_i^1 T \} ,
\]
\[
\hat{M}_i^2 T = \{ q \in Q \mid \exists \sigma_{ed} . (q, \sigma_{ed}, \sigma_s) \not\in \hat{Q} \setminus \hat{M}_i^1 T \} ,
\]
\[
\hat{M}_i^3 T = \{ q \in Q \mid \forall \sigma_{ed} . (q, \sigma_{ed}, \sigma_s) \not\in \hat{Q} \setminus \hat{M}_i^3 T \} .
\]

From Definition 6.1.1

\[
\forall q . \forall \sigma_{ed} . \forall \sigma_s . \forall \sigma_{ep} . (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q} \quad (6.1)
\]

\[
\leftrightarrow (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D .
\]

Then it follows from the definitions of $EX T$ in $G$ and $\hat{M}_i T$ in $\hat{G}$ that:

\[
\forall (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q} .
\]
\[
(q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{M}_i^4 T \leftrightarrow (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in EX T .
\]

One can therefore rewrite $\hat{M}_i^4 T$ as follows:

\[
\hat{M}_i^4 T = \{ q \in Q \mid \forall \sigma_{ed} . \forall \sigma_s . \forall \sigma_{ep} . (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \not\in D \}
\]
\[
\lor (\exists \sigma_s . (\forall \sigma_{ep} . (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \not\in D \setminus EX T))
\]
\[
\lor (\exists \sigma_{ep} . (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D)) \} .
\]
which leads to the conclusion that $MX_1 T = \hat{MX}_1 T$. Since

$$\hat{T}_i = \hat{MX}_1^{0-4} T,$$

and only $\hat{MX}_1 T$ and $\hat{MX}_1^4 T$ contain base states, it is concluded that $\hat{T}_1 = Q \cap \hat{T}_1$. Similarly, it is shown that $\hat{T}_0 = Q \cap \hat{T}_0$. Therefore $\hat{T}_i = Q \cap \hat{T}_i$. \hfill $\square$

**Corollary 6.2.3.** Let $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi})$ be the associate of input-based game $G = (\Sigma, Q, D, \delta, \Pi)$. Let $T \subseteq Q$ be a set of base states. Then

$$\bigcup_{n \geq 0} h^n(\emptyset) = \bigcup_{n \geq 0} g^n(\emptyset),$$

where

$$h(X) = T \lor MX_1 X \quad \text{and}$$

$$g(X) = T \lor \hat{MX}_1^4 X.$$

**Proof.** By induction on $n$. Since $h^0(\emptyset) = g^0(\emptyset) = \emptyset$, the base case holds. Suppose $g^{n-1}(\emptyset) = h^{n-1}(\emptyset)$; thanks to Lemma 6.2.2 it follows that $h^n(\emptyset) = g^n(\emptyset)$. \hfill $\square$

Let $\hat{S} \subseteq \hat{Q}$ be a set of states in $\hat{G}$ and let $\bar{S} \subseteq \hat{S}$ be the set of non-base states such that

$$\bar{S} = \{q \in \hat{S} \mid \gamma(q) \not\in \hat{S}\}.$$

Then $\hat{S}$ is an $i$-based set if

$$\forall q \in \bar{S}. q \in \hat{MX}_1^{1-3}(\hat{S} \setminus \bar{S})$$

Let

$$\check{S} = \{q \in \hat{S} \setminus \bar{S} \mid q \neq \gamma(q)\}.$$ 

Then $\check{S} = \bar{S} \cup (\check{S} \cap Q) \cup \check{S}$. Note that an $i$-closed set (one such that Player $i$ can prevent the play from escaping it) is also $i$-based.

**Lemma 6.2.4.** Let $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi})$ be the associate of input-based game $G = (\Sigma, Q, D, \delta, \Pi)$. Let $\hat{S} \subseteq \hat{Q}$ be a set of states in $\hat{G}$. Then $\gamma(q) \not\in \gamma(\hat{S})$ implies that if $q$ can be attracted to $\hat{S}$ by Player $i$ then $q$ can also be attracted to $\gamma(\hat{S})$ by Player $i$. 


Proof. From Definition 6.1.1 it follows that if there is a path between states \( q \) and \( q' \) in \( \hat{Q} \) such that \( \gamma(q) \neq \gamma(q') \), then that path must visit \( \gamma(q') \). Therefore \( q \) can only be forced to visit any state in \( \hat{S} \) after visiting \( \gamma(\hat{S}) \). Therefore \( q \) is attracted to \( \gamma(\hat{S}) \).

\[\square\]

Lemma 6.2.5. Let \( \hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi}) \) be the associate of input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \). Let \( \hat{T} \subseteq \hat{Q} \) be an \( i \)-based set and \( \hat{P} = Q \cap \hat{T} \). Let \( \text{attr}_i(\hat{G}, \hat{T}) \) and \( \text{attr}_i(\hat{G}, \hat{P}) \). Then \( Q \cap \hat{P}_i = Q \cap \hat{T}_i \).

Proof. Consider \( q \in Q \). If \( q \in \hat{T} \), then \( q \) trivially belongs to both attractions. If, on the other hand, \( q \not\in \hat{T} \), it must be \( \gamma(q) \not\in \gamma(\hat{T}) \). Suppose \( q \in \hat{T}_i \). Then Player \( i \) can force a visit of \( \hat{T} \) from \( q \). By definition of \( i \)-based set, Player \( i \) can force a visit of \( \hat{T} \setminus T \) from every state of \( T \). Therefore, Player \( i \) can force a visit of \( \hat{T} \setminus T \) from \( q \). Since \( \gamma(\hat{T} \setminus T) = \hat{T} \cap Q \), Lemma 6.2.4 implies that \( Q \cap \hat{T}_i \subseteq Q \cap \hat{P}_i \). The inclusion in the opposite direction follows from the monotonicity of attractions.

\[\square\]

The correspondence is now established between \( i \)-attractions in \( G \) and \( \hat{G} \).

Lemma 6.2.6. Let \( \hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi}) \) be the associate of input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \). Let \( \hat{T} \subseteq \hat{Q} \) be an \( i \)-based set and \( T = Q \cap \hat{T} \). Then

\[ T_i = Q \cap \hat{T}_i , \]

where \( \hat{T}_i = \text{attr}_i(\hat{G}, \hat{T}) \) and \( T_i = \text{attr}_i(G, T) \).

Proof. Let \( f(X) = \hat{T} \lor \hat{M}X_i X, \varphi(X) = T \lor \hat{M}X_i X, \) and \( g(X) = T \lor \hat{M}X_i^4 X \). These functions are monotonic over finite lattices; therefore they are continuous. It follows from Corollary 6.2.3:

\[ \hat{T}_i = \bigcup_{n \geq 0} f^n(\emptyset) \quad \text{and} \quad T_i = \bigcup_{n \geq 0} g^n(\emptyset) . \]

Moreover, from Lemma 6.2.5 and the fact that \( \hat{T} \) is \( i \)-based,

\[ Q \cap \bigcup_{n \geq 0} f^n(\emptyset) = Q \cap \bigcup_{n \geq 0} \varphi^n(\emptyset) . \]

While in general \( \hat{M}X_i \) does not distribute over union, \( \hat{G} \) is four-partite, therefore:

\[ \hat{M}X_i(S_1 \cup S_2) = \hat{M}X_i S_1 \cup \hat{M}X_i S_2 \]
if every state in $S_1$ is of different type from every state in $S_2$. Hence,

$$\bigcup_{n \geq 0} \varphi^n(\emptyset) = \bigcup_{n \geq 0, 0 \leq j < 4} \tilde{M}X^j_i g_{\frac{n+3-j}{4}}(\emptyset).$$

This can be rewritten as

$$\bigcup_{n \geq 0} \varphi^n(\emptyset) = \bigcup_{0 \leq j < 4} \bigcup_{n \geq 0} \tilde{M}X^j_i g^n(\emptyset).$$

Only the first of the four components ($j = 0$) has a non-null intersection with the set of base states. Therefore

$$Q \cap \bigcup_{n \geq 0} f^n(\emptyset) = Q \cap \bigcup_{n \geq 0} \varphi^n(\emptyset) = \bigcup_{n \geq 0} g^n(\emptyset).$$

Hence $T_i = Q \cap \hat{T}_i$. \(\square\)

As shown in Fig. 3.7, Zielonka’s algorithm recurs on the subgame $\hat{G}^S$ induced by the removal of $\hat{T}_i$ from $\hat{G}$. In Fig. 6.1, $\hat{T}_0 \neq \hat{T}_1$ but $T_0 = T_1$ which indicates that the subgame $\hat{G}^S$ is not necessarily the associate of the subgame of $G$ obtained by removing $T_i$. Two issues confront us. The first is that the game graph of a subgame $G \setminus T_i$ is not the subgraph induced by $Q \setminus T_i$. The input-based subgame must account for the fact that even when two base states of a turn-based game belong to a subgame, some of the paths connecting them may not be entirely contained in it. In Fig. 6.1, the subgame of $G$ induced by removing $T_0$ must not contain the edge $(q, (1, 1, 1), t)$ even though both $q$ and $t$ belong to the subgame $(G \setminus T_0)$. The following is the proper definition of the induced subgame with respect to the removal of $i$-attraction $T_i$ from $G$.

**Definition 6.2.7.** An input-based game $G^{s_i} = (\Sigma, Q^{s_i}, D^{s_i}, \delta^{s_i}, \Pi^{s_i})$ is a subgame of input-based game $G = (\Sigma, Q, D, \delta, \Pi)$ induced by removing from $G$ an $i$-attraction $T_i$, if

1. $T = T_i \times \Sigma$

2. $Q^{s_i} = Q \setminus T_i$

3. $D^{s_i} = D \setminus (T \cup EXT_1 \cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) | \exists \sigma'_{ed}, \forall \sigma'_{ep}. (q, (\sigma'_{ed}, \sigma'_s, \sigma'_{ep})) \notin (D \setminus EXT_1)\}).$
\[ D^{s_0} = D \setminus (\mathcal{T} \cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ed})) \mid \exists \sigma'_{ep}. (q, (\sigma_{ed}, \sigma_s, \sigma'_{ed})) \in \text{EX}T_0\}) \),

\begin{align*}
\forall u, v \in Q^{s_i}. \forall (u, \sigma) \in D^{s_i}. \\
\delta^{s_i}(u, \sigma) = v \iff \delta(u, \sigma) = v,
\end{align*}

(5) \( \Pi^{s_i} \) is the restriction of \( \Pi \) to \( Q^{s_i} \).

The second issue that needs to be dealt with is that the subgame \( \hat{G}^S \) is not necessarily the associate of any subgame of input-based game \( G \). However, there exists a subgame \( \hat{G}^s \) of \( \hat{G}^S \) that is the associate of the corresponding subgame of \( G \). Let \( B_u \) be the states of \( \hat{G}^S \) unreachable from the base states \( Q \cap \hat{Q}^S \); then \( \hat{G}^s \), the subgame of \( \hat{G}^S \) obtained by removing the states in \( B_u \), is the subgame of \( \hat{G} \) reach-reduced from \( Q^S \).

Lemma 6.2.8 and 6.2.9 prove that \( \hat{G}^s \) is the associate of \( G^S \) when \( \hat{G}^S \) and \( G^S \) are obtained by removing corresponding attractions from associate games.

No play starting in a state in \( \hat{Q}^S \setminus B_u \) visits any state in \( B_u \). Thus, no strategy in \( \hat{G}^s \) relies on states in \( B_u \), and a winning strategy for Player \( i \) in \( \hat{G}^s \) is a winning strategy in \( \hat{G}^S \) for the same player from the same state. The computation of strategies for states in \( B_u \) involves attraction computations. This means that the ability to derive strategies for one game from the strategies of its associate established in Theorem 6.1.7 extends to subgames induced by corresponding \( i \)-attractions.

It is now proven that the subgames encountered during the run of \( \text{ib-Z} \) on an input-based game are the associates of the reach-reduced subgames encountered when playing of the turn-based game.

**Lemma 6.2.8.** Let \( \hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi}) \) be the associate of input-based game \( G = (\Sigma, Q, D, \delta, \Pi) \) and \( \hat{T}_1 \) be \( \text{attr}_1(\hat{G}, \hat{T}) \). If \( \hat{G}^s \) is the subgame of \( \hat{G} \) reach-reduced from \( \hat{Q} \setminus \hat{T}_1 \) then it is the associate of \( G^{s_1} = (\Sigma, Q^{s_1}, D^{s_1}, \delta^{s_1}, \Pi^{s_1}) \), where \( G^{s_1} \) is the subgame of \( G \) induced by \( Q \setminus T_1 \), where \( T_1 = Q \cap \hat{T}_1 \) is the corresponding 1-attraction of \( T = Q \cap \hat{T} \) in \( G \).

**Proof.** The parent game \( \hat{G} \) is the associate game of \( G \), thus the base states of \( \hat{G} \) are the states of \( G \). To prove that \( \hat{G}^s \) is the associate of \( G^{s_1} \), it is first shown that the base states of \( \hat{G}^s \) are the states of \( G^{s_1} \). According to the definition of reach-reduced game, the base states of \( \hat{G}^s \) are \( \hat{Q} \setminus \hat{T}_1 \). Since, \( Q^{s_1} = Q \setminus T_1 \) and \( T_1 = Q \cap \hat{T}_1 \),
\[ \Sigma_{ed} = \{0, 1\} \]
\[ \Sigma_1 = \{0, 1\} \]
\[ \Sigma_{ep} = \{0, 1\} \]
\[ T = T = \{s\} \]

\[ \hat{T}_0 = \{(q, 1, 1, 1), (q, 1, 1), s\} \]
\[ T_0 = \{s\} \]
\[ (q, (1, 0, 1)) \in D^{s_0} \]
\[ (q, (1, 1, 0)) \notin D^{s_0} \]
\[ (q, (1, 1, 1)) \notin D^{s_0} \]

\[ \hat{T}_1 = \{(q, 1, 1, 1), s\} \]
\[ T_1 = \{s\} \]
\[ (q, (1, 0, 1)) \in D^{s_1} \]
\[ (q, (1, 1, 0)) \notin D^{s_1} \]
\[ (q, (1, 1, 1)) \notin D^{s_1} \]

Figure 6.1: The subgame of an input-based game induced by the removal of states in \(i\)-attraction depends on the value of \(i\).

The base states of \(\hat{G}^s\) are the states of \(G^{s_1}\). From Definition 6.1.1 it is known that (6.1) relates the non-base states of the associate game and the set of state and input-label pairs. It is now shown that (6.1) also holds for \(G^{s_1}\) and \(\hat{G}^s\).

It is first shown that

\[ \forall q \cdot \forall \sigma_{ed} \cdot \forall \sigma_s \cdot \forall \sigma_{ep} \cdot (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \hat{Q}^s \]

\[ \rightarrow (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} \cdot \]

(6.2)

Suppose \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \hat{Q}^s\), but \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D^{s_1}\). Since \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \hat{Q}^s\), the base state \(q\) is in \(Q^s\) and therefore \(q \notin T_1\). This implies that

\[ (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin T \cdot \]
From Definition 6.1.1, \( \hat{\delta}((q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \) is a singleton and its element is a base state; this implies that 
\[ \hat{\delta}((q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \subseteq \hat{Q}^s, \]
therefore \( \delta((q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))) \in Q^s \) which means \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \text{EX } Q^{s_1}. \)
Since \( Q^{s_1} \cap T_1 = \emptyset, \)
\[ (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin \text{EX } T_1 . \]
Since \( \hat{G}^s \) is reach-reduced, \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \) is reachable from \( q \) in \( \hat{G}^s \); therefore Player 1 cannot force a visit of \( T_1 \) from \( q \) in \( G \). Since \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D, \) from Definition 6.2.1 it is concluded that 
\[ (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \notin \{ (q, (\sigma_{ed, \sigma_s, \sigma_{ep}}) ) | \exists \sigma_{s}' . \forall \sigma_{ep}' \cdot (q, (\sigma_{ed, \sigma_s', \sigma_{ep}})) \notin (D \setminus \text{EX } T_1) \} . \]
Therefore \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in D^{s_1}, \) the assumption is wrong and (6.2) holds.

It is now shown that 
\[ \forall q . \forall \sigma_{ed} . \forall \sigma_s . \forall \sigma_{ep} . (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in D^{s_1} \]
\[ \rightarrow (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q}^s . \]
Suppose \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in D^{s_1}, \) but \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \notin \hat{Q}^s. \) If the base state \( q \) is not in \( \hat{Q}^s \) then \( q \in \hat{T}_1, \) which implies \( q \in T_1. \) Therefore \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in T. \) Hence \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \notin D^{s_1}. \)

If the base state \( q \) is in \( \hat{Q}^s, \) then either \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \in \hat{T}_1 \) or \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \) is unreachable from any base state in \( \hat{Q}^s. \) If \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \in \hat{T}_1 \) then \( \delta((q, (\sigma_{ed, \sigma_s, \sigma_{ep}}))) \in T_1 \) because \( \hat{\delta}((q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \subseteq T_1 \) is a singleton. Therefore \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in \text{EX } T_1. \) Hence \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \notin D^{s_1}. \)

In the latter case, \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \) is unreachable from any base state in \( \hat{Q}^s \) because \( (q, \sigma_{ed}) \in \hat{T}_1. \)
Since \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in D, \) from Definition 6.2.1 it is concluded that 
\[ (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \in \{ (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) | \exists \sigma_{s}' . \forall \sigma_{ep}' \cdot (q, (\sigma_{ed, \sigma_s', \sigma_{ep}})) \notin (D \setminus \text{EX } T_1) \} . \]
Hence \( (q, (\sigma_{ed, \sigma_s, \sigma_{ep}})) \notin D^{s_1}. \) All the cases contradict the assumption and therefore (6.3) holds. Since both (6.2) and (6.3) are true, it is concluded that (6.1) holds.

From Definition 6.1.1, the set of non-base states in \( \hat{G}^s \) can be determined directly from \( D^{s_1}. \) State \( (q, \sigma_{ed, \sigma_s, \sigma_{ep}}) \) is only reachable from base state \( q \) through \( (q, \sigma_{ed}) \) and then \( (q, \sigma_{ed, \sigma_s}). \) Since (6.1) holds,
the state \((q, \sigma_{ed}, \sigma_s, \sigma_{ep})\) is in \(G^s\) and \(\forall q \in Q^s, \delta^s(q) \neq \emptyset\),

\[
\hat{Q}^s = Q^s \cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} \mid (q, \sigma_{ed})\}
\]
\[
\cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} \mid (q, \sigma_{ed}, \sigma_s)\}
\]
\[
\cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} \mid (q, \sigma_{ed}, \sigma_s, \sigma_{ep})\}.
\]

The states of \(G^s\) and the states of \(G^{s_1}\) are in correspondence with each other according to Definition 6.1.1. Since \(\hat{G}\) is the associate of \(G\), both \(\Pi^{s_1}\) and \(\delta^{s_1}\) are restrictions of \(\Pi\) and \(\delta\) with respect to \(Q^{s_1}\), while both \(\hat{\Pi}^s\) and \(\hat{\delta}^s\) are restrictions of \(\hat{\Pi}\) and \(\hat{\delta}\) with respect to \(\hat{Q}^s\). One can conclude that \(\hat{G}^s\) is the associate of \(G^{s_1}\). 

**Lemma 6.2.9.** Let \(\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\Pi})\) be the associate of input-based game \(G = (\Sigma, Q, D, \delta, \Pi)\) and \(\hat{T}_0\) be attr(\(G, \hat{T}\)). If \(\hat{G}^s\) is the subgame of \(\hat{G}\) reach-reduced from \(\hat{Q} \setminus \hat{T}_0\) then it is the associate of \(G^{s_0} = (\Sigma, Q^{s_0}, D^{s_0}, \delta^{s_0}, \Pi^{s_0})\), where \(G^{s_0}\) is the subgame of \(G\) induced by \(Q \setminus T_0\), where \(T_0 = Q \cap \hat{T}_0\) is the corresponding \(0\)-attraction of \(T = Q \cap \hat{T}\) in \(G\).

**Proof.** The parent game \(\hat{G}\) is the associate game of \(G\), thus the base states of \(\hat{G}\) are the states of \(G\). To prove that \(\hat{G}^s\) is the associate of \(G^{s_0}\), it is first shown that the base states of \(\hat{G}^s\) are the states of \(G^{s_0}\). According to the definition of reach-reduced game, the base states of \(\hat{G}^s\) are \(Q \setminus \hat{T}_0\). Since \(Q = Q \setminus T_0\) and \(T_0 = Q \cap \hat{T}_0\), the base states of \(\hat{G}^s\) are the states of \(G^{s_0}\). From Definition 6.1.1 it is known that (6.1) relates the non-base states of the associate game and the set of state and input-label pairs. It is shown that (6.1) also holds for \(G^{s_0}\) and \(\hat{G}^s\).

First it is shown that

\[
\forall q, \forall \sigma_{ed}, \forall \sigma_s, \forall \sigma_{ep}, (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q}^s
\]

\[
\rightarrow (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_0}.
\]

Suppose \((q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q}^s\), but \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D^{s_0}\). Since \((q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{Q}^s\), the base state \(q\) is in \(Q^s\). Therefore \(q \notin T_0\). This implies that \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin T\).

From Definition 6.1.1, \(\delta((q, \sigma_{ed}, \sigma_s, \sigma_{ep}))\) is a singleton and its element is a base state; this implies that \(\delta((q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \subseteq \hat{Q}^s\), therefore \(\delta((q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))) \in Q^s\) which means \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in EX Q^{s_0}\). Since \(Q^{s_0} \cap T_0 = \emptyset\), \((q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin EX T_0\).
Since \( \hat{G}^s \) is reach-reduced, \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \) is reachable from \( q \) in \( \hat{G}^s \); therefore Player 0 cannot force a visit of \( T_0 \) from \( q \) in \( G \). Since \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D \), from Definition 6.2.1 it is concluded that

\[
(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) | \exists \sigma'_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma'_{ed})) \in EX T_0\}.
\]

Therefore \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_0} \), the assumption is wrong and (6.4) holds.

It is now shown that

\[
\forall q, \forall \sigma_{ed}, \forall \sigma_s, \forall \sigma_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} \rightarrow (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{G}^s.
\]

Suppose \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_0} \), but \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \notin \hat{Q}^s \). If the base state \( q \) is not in \( \hat{Q}^s \) then \( q \in \hat{T}_0 \), which implies that \( q \in T_0 \). Therefore \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in T \). Hence \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D^{s_0} \).

If the base state \( q \in \hat{Q}^s \), then either \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{T}_0 \) or \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \) is unreachable from any base state in \( \hat{Q}^s \). If \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \in \hat{T}_0 \) then \( \delta((q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))) \in T_0 \) because \( \delta((q, \sigma_{ed}, \sigma_s, \sigma_{ep})) \subseteq T_0 \) is a singleton. Therefore \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in EX T_0 \). Hence \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D^{s_0} \).

In the latter case, \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \) is unreachable from any base state in \( \hat{Q}^s \) because \( (q, \sigma_{ed}, \sigma_s) \in \hat{T}_0 \). Since \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D \), from Definition 6.2.1 it is concluded that

\[
(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) | \exists \sigma'_{ep}, (q, (\sigma_{ed}, \sigma_s, \sigma'_{ed})) \in EX T_0\}.
\]

Hence \( (q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \notin D^{s_0} \). All the cases contradict the assumption and therefore (6.4) holds. Since both (6.4) and (6.5) are true, it is concluded that (6.1) holds.

From Definition 6.1.1, the set of non-base states in \( \hat{G}^s \) can be determined directly from \( D^{s_0} \). State \( (q, \sigma_{ed}, \sigma_s, \sigma_{ep}) \) is only reachable from base state \( q \) through \( (q, \sigma_{ed}) \) and then \( (q, \sigma_{ed}, \sigma_s) \). Since \( \forall q \in \hat{Q}^s, \delta^s(q) \neq \emptyset \),

\[
\hat{Q}^s = Q^s \cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} | (q, \sigma_{ed})\}
\]

\[
\cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} | (q, \sigma_{ed}, \sigma_s)\}
\]

\[
\cup \{(q, (\sigma_{ed}, \sigma_s, \sigma_{ep})) \in D^{s_1} | (q, \sigma_{ed}, \sigma_s, \sigma_{ep})\}.
\]
The states of $\hat{G}^s$ and the states of $G^{s_0}$ are in correspondence with each other according to Definition 6.1.1. Since $\hat{G}$ is the associate of $G$, both $\Pi^{s_0}$ and $\delta^{s_0}$ are restrictions of $\Pi$ and $\delta$ with respect to $Q^{s_0}$, while both $\hat{\Pi}^s$ and $\hat{\delta}^s$ are restrictions of $\hat{\Pi}$ and $\hat{\delta}$ with respect to $\hat{Q}^s$. One can conclude that $\hat{G}^s$ is the associate of $G^{s_0}$. \hfill $\Box$

The algorithm of ib-Z is obtained from Zielonka’s algorithm in Figure 3.7 by replacing Definition 3.9.1 with Definition 6.2.1 and Definition 3.9.2 with Definition 6.2.7. It is now proven that the adapted algorithm correctly computes Player $i$’s winning and losing states.

**Theorem 6.2.10.** For an input-based game $G = (\Sigma, Q, D, \delta, \pi)$ ib-Z returns $W_0 = Q \cap \hat{W}_0$ and $W_1 = Q \cap \hat{W}_1$ where $\hat{W}_0$ and $\hat{W}_1$ are the set of states returned by Zielonka’s algorithm for the associate turn-based game $\hat{G} = (\hat{Q}, \hat{Q}_0, \hat{Q}_1, \hat{\delta}, \hat{\pi})$.

**Proof.** It is proven by induction that ib-Z returns Player $i$’s winning and losing base states. If $G$ is an empty game then $\hat{G}$ is also empty; then, on Line 2, ib-Z returns $W_i = W_j = \emptyset$ in agreement with the result of Zielonka’s algorithm on $\hat{G}$. Therefore the base case holds trivially.

If $G$ is a non-empty game then so is $\hat{G}$. Both ib-Z and Zielonka’s algorithm recur on subgames of $G$ and $\hat{G}$ respectively. It is now shown the correspondence between the two subgames. At Line 3 let $T \subseteq Q$ be the set of states colored $k$ with respect to $\pi$ and $\hat{T} \subseteq \hat{Q}$ be the set of states colored $k$ with respect to $\hat{\pi}$. From Definition 6.1.1 it is known that $T = Q \cap \hat{T}$. The set of states $\hat{T}$ is $i$-based because $T = \emptyset$. At Line 5 let $T_i$ be the $i$-attraction of $T$ in $G$ and $\hat{T}_i$ be the $i$-attraction of $\hat{T}$ in $\hat{G}$. Thanks to Lemma 6.2.6, $T_i = Q \cap \hat{T}_i$. At Line 6 ib-Z recurs on the subgame $G^{s_i} = (\Sigma, Q^{s_i}, D^{s_i}, \delta^{s_i}, \pi^{s_i})$ where $Q^{s_i} = Q \setminus T_i$. Since $T_i = Q \cap \hat{T}_i$ and thanks to Lemmas 6.2.8 and 6.2.9, the reach-reduced subgame $\hat{G}^s = (\hat{Q}^s, \hat{Q}_0^s, \hat{Q}_1^s, \hat{\delta}^s, \hat{\pi}^s)$ induced by $\hat{Q} \setminus \hat{T}_i$ is the associate of $G^{s_i}$. On the other hand, Zielonka’s algorithm recurs on the subgame $\hat{G}^S = (\hat{Q}^S, \hat{Q}_0^S, \hat{Q}_1^S, \hat{\delta}^S, \hat{\pi}^S)$ induced by $\hat{Q} \setminus \hat{T}_i$. Since Player $i$’s winning and losing base states in $\hat{G}^s$ are the same as the winning and losing base states in $\hat{G}^S$, the inductive hypothesis can be invoked and it is concluded that at Line 7, $U_i = Q \cap \hat{U}_i$ and $U_j = Q \cap \hat{U}_j$.

If at Line 7 $U_j = \emptyset$ then $Q \cap \hat{U}_j = \emptyset$. Since every path through a non-base state must visit some base state, it follows that $Q \cap \hat{U}_j = \emptyset$ implies $\hat{U}_j = \emptyset$. Therefore the Line 7 evaluates to true in Zielonka’s
algorithm as well. Hence, all states of both $G$ and $\hat{G}$ are winning for Player $i$. ib-Z returns $W_i = Q = Q \cap \hat{Q}$ and $W_j = Q \cap \hat{Q} = \emptyset$ in agreement with the results of Zielonka’s algorithm on $\hat{G}$.

If at Line 7 $U_j \neq \emptyset$ then $\hat{U}_j \neq \emptyset$ because it has been established $U_j = Q \cap \hat{U}_j$. The set $\hat{U}_j$ is $j$-based because $\hat{U}_j$ is $j$-closed in $\hat{G}$. At Line 12, with argument similar to the one for Line 6, the inductive hypothesis can again be invoked, hence $U_i = Q \cap \hat{U}_i$ and $U_j = Q \cap \hat{U}_j$. At Line 13 ib-Z assigns all the base states in $Q \setminus U_i$ to Player $j$ in agreement with Zielonka’s algorithm. Therefore $U_i = Q \cap \hat{U}_i$ and $U_j = Q \cap \hat{U}_j$ is true after Line 13.

At Line 17 ib-Z returns $W_i = Q \cap \hat{W}_i = Q \cap \hat{U}_i$ and $W_j = Q \cap \hat{W}_j = Q \cap \hat{U}_j$. Therefore ib-Z correctly identifies the set of Player $i$’s winning and losing states for $G$. □

By extending Lemma 6.2.6 and Theorem 6.2.10 it can now be shown that the strategies for both players computed by ib-Z are related to those returned by Zielonka’s algorithm through Definition 6.1.2.

It has been shown that Zielonka’s algorithm can be adapted to solve input-based games directly. Since the Chatterjee’s algorithm for conjunctive parity games is a generalization of Zielonka’s algorithm, it stands to reason that Chatterjee’s algorithm can be adapted to play input-based games directly by making similar changes. The advantages of playing input-based games are evident from the results discussed in Chapter 8.

6.3 Mealy Coloring Functions

In the previous sections, the discussion has focused on state based coloring functions. This discussion focuses on edge coloring functions. Consider the property $G(a \to X F b)$. The Figure 6.2 shows two parity automata for this property. The one on the left has state coloring function while one on the right has an edge coloring function. As shown in this example, the advantage of an edge coloring function is that in certain cases the automaton can be simplified.

Consider the input-based game $G = (\Sigma, Q, D, \delta, \pi)$, where $\pi : D \to [k]$ and $[k] = \{i \mid 0 \leq i < k\}$. Suppose $D_k \subseteq D$ is the set of state and input-label pairs that are colored $k$. Then the states from which Player $i$ can force the selection of transitions that are colored $k$ is given by $MX_i D_k$. Before one computes the attractor of states in $MX_i D_k$ one needs to compute the subgame in which the Player $j$ cannot enable
Figure 6.2: The automata for $G(a \rightarrow X F b)$. In automaton on the left, the color of the states are mentioned under the name of the states. In the automaton on the right, all the edges are colored 1 except the one labeled with a circle which is colored 0.

Player $i$ in selecting a transition that is colored $k$. This subgame can be computed by Definition 6.2.7. After incorporating these changes in the algorithm for solving input-based games one can compute the winning and losing states of $G$. 
Chapter 7

Specifications that are not General Reactivity (1)

It has been stated throughout this thesis that in practice the specification of reactive system is a set of properties describing its behavior. In this chapter, three examples of reactive systems are discussed. One of them is a GR(1) specification while the other two are GR(2) specifications. (All these examples can easily be extended to GR(n) specifications.) The objective of this chapter is to demonstrate that in practice there exist reactive systems whose specification is naturally written in such a way that it is not a GR(1) specification. Therefore, the ability to solve efficiently a GR(n) game for $n > 1$ is of practical relevance.

ARM’s Advanced Microcontroller Bus Architecture (AMBA) is one of the on-chip communication standards [AL]. In a system on chip (SoC) this standard is used to connect various blocks such as processor cores, cache memory, and DMA controllers. The simplest on-chip communication architecture is the shared-bus architecture. All the devices that need to communicate are connected to a shared bus. Only one device can perform data transfers on the shared bus at any given time. The bus is accessed in a time-sharing manner by those devices that can initiate a data transfer (masters). The shared-bus architecture limits the parallelism and achievable performance of the SoC, which makes it unsuitable for most highly multi-processing SoC applications. Consequently, the single shared-bus architecture is not scalable to meet the demands of multi-processor systems on chip (MPSoC) [Gia13].

A crossbar switch (also known as full-bus matrix), is a generalization of the shared-bus architecture. Each master device of the system is connected to all the devices that cannot initiate a data transfer (slaves). This topology drastically increases the amount of traffic that can occur simultaneously. For instance, two processing cores can access two separate memory banks concurrently.
The AMBA shared-bus architecture has been specified as a GR(1) reactive system in [BGJ+07a]. Since then a few more examples of GR(1) reactive systems have been published (such as IBM General Buffer Controller [BGJ+07b] and an improved version of AMBA shared-bus architecture [GCH11]). However, all of these examples describe a shared-bus architecture. In the next section a simplified\(^1\) version based on the AMBA Bus specification of [BGJ+07a] is presented. In section 7.2 and 7.3 the AMBA shared-bus architecture is generalized to two different types of crossbar configurations.

### 7.1 Simplified AMBA: Shared-Bus Architecture

Consider the example of four masters that need to access a memory and a bus arbiter. A master either wants to access one word from the memory or wants to access an arbitrary large chunk of memory. In the latter case, the master requests a **locked** transfer. During the locked transfer the master which has been granted access to the memory by the bus arbiter has unrestricted access to it until it relinquishes the control of the bus. On the other hand, if the bus arbiter grants a master the use of the bus for reading a single word, then the master has control of the bus until the memory responds to its request.

Let \( \text{MASTER}_i \) denote the \( i \)th master; it has two input signals \( \{ \text{hready} , \text{hgrant}_i \} \) and two output signals \( \{ \text{hbusreq}_i , \text{hlock}_i \} \). The input data bus, the output data bus and the address bus are not involved in the communication protocol. \( \text{MASTER}_i \) makes its request by asserting the \( \text{hbusreq}_i \) signal. The memory is ready to receive a new request if it asserts the \( \text{hready} \) signal. At this point the arbiter’s response depends on the type of access that was granted to \( \text{MASTER}_j \), which had access to the bus; whether it was granted access to read a single word from memory or read an arbitrary chunk of memory. In the case when the access was to read an arbitrary chunk of memory then the arbiter has to make sure that the master is done with its transfer; \( \text{MASTER}_j \) does that by de-asserting the \( \text{hbusreq}_j \) signal. The arbiter grants access to \( \text{MASTER}_i \) by asserting the \( \text{hgrant}_i \) signal. The connections between the arbiter, the memory and various masters have been depicted in the Figure 7.1.

The LTL specification of the arbiter employed in the shared-bus architecture described above is shown in Figure 7.1. The signal \( \text{busreq} \) is the output of a multiplexer; this multiplexer is controlled by the

\(^1\) For instance, the burst mode has been excluded.
\[ h_i = hbusreq_i, \]
\[ l_i = hlock_i. \]

**Figure 7.1:** Shared-bus architecture between four masters and a single memory unit. Each master can also read the hready signal from the memory. That connection is not shown here.

The master signal which is an output of the arbiter. The master signal identifies the MASTER being granted the access to the bus. The hmastlock signal is asserted if the master being granted the access has requested to read a chunk of memory. The start signal, which is internal to the arbiter, indicates when the arbiter is allowed to grant access to a different MASTER.

The arbiter is the system to be synthesized while the masters and the memory form the environment. Therefore, signals from the masters \{hbusreq_0, hbusreq_1, hbusreq_2, hbusreq_3, hlock_0, hlock_1, hlock_2, hlock_3\} and the signal \{hready\} from the memory are all controlled by the environment. The input signals hbusreq_i and hlock_i are controlled by MASTER_i where hbusreq_i is asserted when the master is requesting access to the memory, while hlock_i is asserted if the master wants to read a chunk of memory; otherwise it is de-asserted. The memory asserts hready when it is ready to receive an-
other request. The signals \{hgrant_0, hgrant_1, hgrant_2, hgrant_3, master, hmastlock, start\} are controlled by the arbiter. The signal hgrant_i is asserted when MASTER_i is granted access to the memory. As stated earlier, the master signal is used to control the output busreq of a multiplexer. (The data and address buses from each master are also controlled by the same master signal.) The hmastlock signal indicates to the MASTERS if a locked access has been granted or not. (MASTER_i may have de-asserted hlock_i when hbusreq_i was asserted transfer but changed its mind about the unlocked transfer request and asserted hlock_i along with hbusreq_i in the next clock cycle. However, if the arbiter had made its mind to grant MASTER_i then it need not grant it a locked request. This is the reason why the arbiter needs to let the master know about the type of access being granted to it through hmastlock.)

Once MASTER_i reads that hgrant_i has been asserted, it starts the read/write sequence. It then has to wait for hready to be asserted again to indicate that a single data transfer has occurred successfully. If MASTER_i had requested to write to the memory then hready from the memory implies that it has written the word to a particular address. If it was a read request from MASTER_i then hready from memory implies that the requested word from the memory is available on its data lines. If the MASTER_i had requested a locked transfer then it can read/write as many words as it wants until it is ready to relinquish the bus. After each data transfer hready is asserted by the memory.

An informal description of the properties in Table 7.1 is now presented. The environment’s safety properties:

(1) All the masters are not making any requests and the memory is ready to receive a request. This is the initial state of the environment.

(2) MASTER_i cannot assert hlock_i unless hbusreq_i is also asserted. In other words, a master cannot request a locked access to memory without making a proper request through hbusreq_i.

(3) The signal busreq is the output of a multiplexer that selects between \{hbusreq_0, hbusreq_1, hbusreq_2, hbusreq_3\} based on the master that has been granted the access to the bus. (This signal is not shown in the Figure 7.1 as its presence is to simplify the presentation of the LTL specification.)
Table 7.1: Shared-bus architecture between four masters and a single slave.

**ENVIRONMENT’S SAFETY PROPERTIES**

1. \( (\bigwedge_{0 \leq i \leq 3} \neg \text{lock}_i \land \neg \text{busreq}_i) \land \text{hready} \)

2. \( \bigwedge_{0 \leq i \leq 3} G(\text{lock}_i \rightarrow \text{busreq}_i) \)

3. \( \bigwedge_{0 \leq i \leq 3} G(\text{master} = i \rightarrow (\text{busreq}_i \leftrightarrow \text{busreq})) \)

**ENVIRONMENT’S LIVENESS PROPERTIES**

1. \( G F(\text{hready}) \)

2. \( G(\text{hmastlock} \rightarrow XF \neg \text{busreq}) \)

**SYSTEM’S SAFETY PROPERTIES**

1. \( \neg \text{hmastlock} \land \text{master} = 0 \land \text{hgrant}_0 \land \bigwedge_{1 \leq i \leq 3} \neg \text{hgrant}_i \)

2. \( \bigwedge_{0 \leq i \leq 3} G(\text{hready} \rightarrow \text{hgrant}_i \leftrightarrow X(\text{master} = i)) \)

3. \( \bigwedge_{1 \leq i \leq 3} G(\neg \text{hgrant}_i \rightarrow (\neg \text{hgrant}_i \land \text{busreq})) \)

4. \( \bigwedge_{0 \leq i \leq 3} G((\text{hready} \land \text{hgrant}_i \land X \text{start}) \rightarrow (\text{lock}_i \leftrightarrow X \text{hmastlock})) \)

5. \( G(\neg \text{hready} \rightarrow X \neg \text{start}) \)

6. \( G(X(\neg \text{start}) \rightarrow (\text{master} \leftrightarrow X \text{master})) \)

7. \( G(X(\neg \text{start}) \rightarrow (\text{hmastlock} \leftrightarrow X \text{hmastlock})) \)

**SYSTEM’S LIVENESS PROPERTIES**

1. \( \bigwedge_{0 \leq i \leq 3} GF(\neg \text{busreq}_i \lor \text{master} = i) \)

The environment’s liveness properties:

(1) The memory will eventually respond to the current request of the master and be ready for the next request.

(2) When \text{MASTER}_i is granted access to the bus so that it can read a chunk of memory then it will eventually relinquish the bus by de-asserting \text{busreq}_i.

The system’s safety properties:

(1) Initially \text{MASTER}_0 is granted unlocked access to the bus and \text{hgrant}_0 is asserted.

(2) It is always the case that one of the masters has the access to the bus. When \text{hready} is de-asserted then the bus is idle however the memory is not in a state to accept more requests.

(3) Apart from \text{MASTER}_0, all the other masters will not be given access to the bus without making a
persistent request. (One of the masters will always have control of the bus.)

(4) If the memory is ready for the next request then the arbiter can choose to grant either a locked access or an unlocked access to the bus to a master. However, if the chosen master has requested a locked access then the arbiter has to grant a locked access.

(5) If the memory is in the middle of responding a single or locked request then a different master cannot be given access to the bus.

(6) If the arbiter has granted access to $\text{MASTER}_i$ and this master has requested a locked transfer then the arbiter will not give access to another master until this master withdraws its request by de-asserting $\text{hbusreq}_i$.

The system’s liveness properties ensure that if a master makes a persistent request then it will eventually be granted access to the bus. Let the $\mathcal{S}_e$ be the conjunction of the environment’s safety properties and $\mathcal{L}_e$ be the conjunction of the environment’s liveness properties. Similarly, let $\mathcal{S}_s$ be the conjunction of the system’s safety properties and $\mathcal{L}_s$ be the conjunction of the system’s liveness properties. Then the GR(1) specification for this example is $(\mathcal{S}_e \rightarrow \mathcal{S}_s) \land (\mathcal{L}_e \rightarrow \mathcal{L}_s)$. (In the examples considered in this chapter the environment is well-separated\(^2\).)

### 7.2 Simplified AMBA: Complete Crossbar Architecture

The simplest example of a crossbar architecture is described in this section, which is of a complete crossbar architecture. Consider a memory that has two identical ports that can both be used for either reads or writes. Such a memory can respond to two requests simultaneously. The task of avoiding race conditions between reads and writes is being handled at a higher layer (i.e., firmware, processing unit, operating system, or even application software). Therefore, the specification being considered here need not be concerned with such race conditions.

The specification of the environment has properties describing the behavior of two multiplexer outputs $\text{busreq}_0$ and $\text{busreq}_1$ because there are going to be two parallel connections to the memory. Similarly,

\(^2\) The concept of an environment being well-separate has been discussed in Section 4.5.
each memory port has its own signal \( h\text{ready}_0 \) and \( h\text{ready}_1 \) to indicate when memory is ready to receive another request on a particular port. The liveness properties state that both ports of the memory will always respond to respective requests and be ready for the next requests. Similarly, the masters being given locked access to the memory will always relinquish control of the bus after some time.

Since the masters only have a single port (they can only make one request at a given time) a \texttt{MASTER}
that has been granted access to one of the ports of the memory cannot be granted access to the other port of the memory (as only one port of the memory will be servicing the request). The two multiplexers are controlled by two separate signals \texttt{master}_0 and \texttt{master}_1. Each port to the memory can support either locked or unlocked access, so there are two signals \texttt{hmastlock}_0 and \texttt{hmastlock}_1 to indicate if a particular request has been granted a locked or unlocked access on the respective port. The \texttt{start}_0 and \texttt{start}_1 signals are used to keep track of when the arbiter can grant access to a different master on the respective port.

Let $S_e$ be the conjunction of all the environment’s safety properties and $L^a_e$ be the conjunction of environment’s liveness properties marked $a$ and $L^b_e$ be the conjunction of environment’s liveness properties marked $b$ in Table 7.3. Let $S_s$ be the conjunction of the system’s safety properties and $L_s$ be the conjunction of the system’s liveness properties. Then the GR1 specification is $(S_e \rightarrow S_s) \land (L^a_e \land L^b_e \rightarrow L_s)$. According to this specification if one of the ports of the memory refuses to respond to a request or one of the masters refuses to relinquish the bus after being granted access to it, then the arbiter need not let any requests be serviced on the other port.

On the other hand, if the specification described by the formula $(S_e \rightarrow S_s) \land (L^a_e \rightarrow L_s) \land (L^b_e \rightarrow L_s)$ is considered, then even if the memory stops responding on one port or after being granted locked access the \texttt{MASTER}_i does not want to relinquish the control of the bus connected to this port, the arbiter must continue to service the requests (if it was \texttt{MASTER}_i that was misbehaving then only the requests of remaining masters need to be serviced) as long as the memory keeps responding to requests on that port and no other master misbehaves. However, this specification can be manipulated to represent a GR(1) specification (GR($n$) specifications have been defined in Section 2.6.1). The explanation in Section 7.4 provides clues regarding its reduction to a GR(1) specification.

### 7.3 Simplified AMBA: Partial Crossbar Architecture

Suppose the memory access pattern of the masters is such that a complete crossbar architecture is not required (this happens when a compromise between silicon area and performance is possible). In this particular example the masters \texttt{MASTER}_0 and \texttt{MASTER}_3 are tied to Port-0 of the memory while \texttt{MASTER}_1
is tied to Port-1 of the memory. The requests of MASTER$_2$ are more important, hence it is not tied to either port of the memory; however, it will only be given access to memory through Port-0 if it was requesting an unlocked access to the memory, while Port-1 can respond to both locked and unlocked requests. Notice that in Table 7.3, system’s property 7 states that if hgrant$_2$ is asserted and the hlock$_2$ was also asserted then the transfer must be through Port 1 (as the signal master$_1$ is tied to the communication on Port 1).

Let $S_e$ be the conjunction of all the environment’s safety properties, $L^a_e$ the conjunction of the environment’s liveness properties marked $a$, and $L^b_e$ the conjunction of the environment’s liveness properties marked $b$. Let $S_s$ be the conjunction of the system’s safety properties, $L^a_s = \{1, 3a, 4\}$ the conjunction of the system’s liveness properties for the control of Port 0, and $L^b_s = \{2, 3b\}$ the conjunction of the system’s liveness properties for the control of Port 1. Then the GR1 specification is $(S_e \rightarrow S_s) \land (L^a_e \land L^b_e \rightarrow L^a_s \land L^b_s)$.

As was the case earlier, if one of the ports of the memory refuses to respond to a request or one of the masters refuses to relinquish the bus after being granted access to it then the arbiter need not let any requests be serviced on the other port.

On the other hand, if the specification described by the formula $(S_e \rightarrow S_s) \land (L^a_e \rightarrow L^a_s) \land (L^b_e \rightarrow L^b_s)$ is considered, then the arbiter must continue to service the requests on the other port as long as the memory keeps responding to requests and the remaining masters keep relinquishing the control of the bus after being granted a locked access. This specification is not a GR(1) specification; rather it is a GR(2) specification. Notice that if the request from MASTER$_2$ is for a locked transfer then the arbiter is free to ignore the request from this master on Port-0. This is reflected in system’s liveness property (3a). If both $L^a_s$ and $L^b_s$ contained (3b) then the specification would be unrealizable as the environment can block Port-1 (it can do this by either making MASTER$_1$ misbehave or by making Port-1 unresponsive) and forcing MASTER$_2$ to request for a locked transfer. The system is unable to grant access to MASTER$_2$ on Port-0 while Port-1 has been blocked. Notice that through this strategy the environment lets the system satisfy the property $(L^b_e \rightarrow L^b_s)$ as the environment violates $L^b_e$. However, the system has lost the ability to satisfy $(L^a_e \rightarrow L^a_s)$ even when the environment satisfies $L^a_s$ (the set of safety properties $S_s$ does not allow the system to satisfy $L^a_s$).

Sections 7.2 and 7.3 discuss two simple examples of a crossbar architecture. The obvious LTL specifications of both these example belong to the class of GR(2) properties. However, the specification of the
Table 7.3: Crossbar architecture between four masters and a single slave with two identical ports. MASTER\(_0\) and MASTER\(_3\) can only read through Port-0 of the memory while MASTER\(_1\) can only read through Port-1 of the memory. If MASTER\(_2\) is requesting a locked access then it will only be given such an access through Port-1.

### Environment’s Safety Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(\bigwedge_{0\leq i \leq 3} \neg \text{lock}_i \land \neg \text{busreq}_i \land \text{hready}_0 \land \text{hready}_1)</td>
</tr>
<tr>
<td>2.</td>
<td>(\bigwedge_{0 \leq i \leq 3} G(h\text{lock}_i \rightarrow h\text{busreq}_i))</td>
</tr>
<tr>
<td>3a.</td>
<td>(\bigwedge_{i \in {0, 2, 3}} G(\text{master}_0 = i \rightarrow (h\text{busreq}_i \leftrightarrow \text{busreq}_0)))</td>
</tr>
<tr>
<td>3b.</td>
<td>(\bigwedge_{1 \leq i \leq 2} G(\text{master}_1 = i \rightarrow (h\text{busreq}_i \leftrightarrow \text{busreq}_1)))</td>
</tr>
</tbody>
</table>

### Environment’s Liveness Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a.</td>
<td>GF(\text{hready}_0)</td>
</tr>
<tr>
<td>1b.</td>
<td>GF(\text{hready}_1)</td>
</tr>
<tr>
<td>2a.</td>
<td>G(h\text{mastlock}_0 \rightarrow XF \neg \text{busreq}_0)</td>
</tr>
<tr>
<td>2b.</td>
<td>G(h\text{mastlock}_1 \rightarrow XF \neg \text{busreq}_1)</td>
</tr>
</tbody>
</table>

### System’s Safety Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>master(_0) = 0 \land master(<em>1) = 1 \land \bigwedge</em>{0 \leq j \leq 1} \neg \text{hmastlock}_j \land \text{hgrant}<em>j \land \bigwedge</em>{2 \leq i \leq 3} \neg \text{hgrant}_i</td>
</tr>
<tr>
<td>2.</td>
<td>G(h\text{ready}_0 \rightarrow h\text{grant}_0 \leftrightarrow \text{master}_0 = 0))</td>
</tr>
<tr>
<td></td>
<td>G(h\text{ready}_0 \rightarrow h\text{grant}_2 \leftrightarrow \text{master}_0 = 2 \oplus \text{master}_1 = 2))</td>
</tr>
<tr>
<td></td>
<td>G(h\text{ready}_0 \rightarrow h\text{grant}_3 \leftrightarrow \text{master}_0 = 3))</td>
</tr>
<tr>
<td></td>
<td>G(h\text{ready}_1 \rightarrow h\text{grant}_1 \leftrightarrow \text{master}_1 = 1))</td>
</tr>
<tr>
<td>3.</td>
<td>(\bigwedge_{2 \leq i \leq 3} G(\neg \text{hgrant}_i \rightarrow (\neg \text{hgrant}_i \wedge \neg \text{busreq}_i)))</td>
</tr>
<tr>
<td>4a.</td>
<td>(\bigwedge_{i \in {0, 2, 3}} G((h\text{ready}_0 \land h\text{grant}_0 \land X \text{start}_0 \land X \text{master}_0 = 0) \rightarrow (\neg \text{lock}_0 \leftrightarrow X h\text{mastlock}_0)))</td>
</tr>
<tr>
<td>4b.</td>
<td>(\bigwedge_{2 \leq i \leq 3} G((h\text{ready}_1 \land h\text{grant}_1 \land X \text{start}_1 \land X \text{master}_1 = 1) \rightarrow (\neg \text{lock}_1 \leftrightarrow X h\text{mastlock}_1)))</td>
</tr>
<tr>
<td>5a.</td>
<td>G(\text{hready}_0 \rightarrow X \neg \text{start}_0)</td>
</tr>
<tr>
<td></td>
<td>G(X(\neg \text{start}_0) \rightarrow (\text{master}_0 \leftrightarrow X \text{master}_0))</td>
</tr>
<tr>
<td>5b.</td>
<td>G(X(\neg \text{start}_0) \rightarrow (h\text{mastlock}_0 \leftrightarrow X h\text{mastlock}_0))</td>
</tr>
<tr>
<td></td>
<td>G(\text{hready}_1 \rightarrow X \neg \text{start}_1)</td>
</tr>
<tr>
<td></td>
<td>G(X(\neg \text{start}_1) \rightarrow (\text{master}_1 \leftrightarrow X \text{master}_1))</td>
</tr>
<tr>
<td></td>
<td>G(X(\neg \text{start}_1) \rightarrow (h\text{mastlock}_1 \leftrightarrow X h\text{mastlock}_1))</td>
</tr>
<tr>
<td>6a.</td>
<td>G(h\text{mastlock}_0 \land \text{start}_0 \rightarrow (\neg \text{start}_0 W(\neg \text{start}_0 \land \neg \text{busreq}_0)))</td>
</tr>
<tr>
<td>6b.</td>
<td>G(h\text{mastlock}_1 \land \text{start}_1 \rightarrow (\neg \text{start}_1 W(\neg \text{start}_1 \land \neg \text{busreq}_1)))</td>
</tr>
<tr>
<td>7.</td>
<td>G((h\text{ready} \land h\text{grant}_2 \land h\text{lock}_2) \rightarrow X \text{master}_1 = 2)</td>
</tr>
</tbody>
</table>

### System’s Liveness Properties

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>GF(\neg \text{busreq}_0 \lor \text{master}_0 = 0)</td>
</tr>
<tr>
<td>2.</td>
<td>GF(\neg \text{busreq}_1 \lor \text{master}_1 = 1)</td>
</tr>
<tr>
<td>3a.</td>
<td>GF(\neg \text{busreq}_2 \lor \neg \text{lock}_2 \lor \text{master}_0 = 2 \lor \text{master}_1 = 2)</td>
</tr>
<tr>
<td>3b.</td>
<td>GF(\neg \text{busreq}_2 \lor \text{master}_0 = 2 \lor \text{master}_1 = 2)</td>
</tr>
<tr>
<td>4.</td>
<td>GF(\neg \text{busreq}_3 \lor \text{master}_0 = 3)</td>
</tr>
</tbody>
</table>
example in Section 7.2 can be reduced to a GR(1) specification, the specification of the example in Section 7.3 cannot be reduced to a GR(1) specification. A GR(2) specification can be converted to a Streett(2) game. (The conversion of a GR(1) specification to a Streett(1) game has been discussed in Section 4.3.1. This process can be extended to convert a GR(2) specification to a Streett(2) game.)

When one needs to play a game obtained from the GR(2) specification then one can reduce such a game to a safety game through the procedure of [BJW02], however, such a reduction yields a safety game that is significantly larger. Furthermore, finding an efficient implementation from the winning strategies of such a game is not trivial. An efficient algorithm to solve the Semi-Generalized Streett(2) game obtained from a GR(2) specification has been discussed in Section 4.4, which exploits the special structure of such a game.

One may wonder why a GR(2) specification is necessary; this is now explained using the Serial attached SCSI (SAS) controller (SCSI stands for Small Computer System Interface). Various number of masters request the SAS controller for access to many SCSI hard drives (the hard drives may only have a single read/write port or a two read/write ports). A master requests for a connection to one of the hard drives and after permission is granted it proceeds to read/write access to the hard drive; when it completes its transaction it closes the connection. The SAS controller may allow various number of simultaneous connections (depending on the number of masters and number of hard drives) to increase the efficiency of the information exchange between the masters and the hard drives. The SAS controller must keep on serving requests of different masters even if one of the hard drives or masters misbehave and block one of the connections (for instance one of the memory ports in the example of Section 7.3 never asserts the \texttt{hready}_1 again). If such a controller is specified through a GR(1) specification then the controller is not obligated to continue serving requests from different masters when one of the masters or hard-drive misbehave even when the controller has the ability to serve requests in limited capacity. However in practice, the desire is for the controller to continue serving requests as long as some of its connections are unblocked.
7.4 Reduction of General Reactive (2) to General Reactive (1)

Even when the specification appears to satisfy the syntactic characterization of a GR(n) formula, one wonders if such a specification can be reduced to some other GR(n - k) formula. The discussion in this section highlights the simple mechanism in which one may be able to answer such a question. Consider the following two simple reactive formulae

\[(G F h_0 \rightarrow GF p \land GF q \land GF r) \land (G F h_1 \rightarrow GF p \land GF q \land GF r)\]

\[(7.1)\]

\[(G F h_0 \rightarrow GF p \land GF q) \land (G F h_1 \rightarrow GF q \land GF r)\]

\[(7.2)\]

one of which is a GR(1) formula while the other does not have a corresponding GR(1) formula. Algebraic manipulation of (7.1) and some knowledge of LTL show that it can be rewritten as

\[GF(h_0 \lor h_1) \rightarrow GF p \land GF q \land GF r\]

which is clearly a GR(1) formula. On the other hand, no algebraic manipulation can convert (7.2) into a GR(1) formula because no equivalent GR(1) formula exists as discussed in the remainder of this section.

Consider the set of propositions \(L = \{h_0, h_1, p, q, r\}\) and an alphabet \(\Sigma_L\) derived from it. This alphabet has 32 letters which are the unique subsets of \(L\). Construct a deterministic automaton with 32 states with each state being labeled with a unique letter from \(\Sigma_L\) while every state has a transition into every other state. The acceptance condition of this automaton is provided by (7.2), which can be a Streett condition with four pairs \(\{(L_1, U_1), (L_1, U_2), (L_2, U_2), (L_2, U_3)\}\) where \(L_1\) is the set of states whose label contains the proposition \(h_0\) and \(L_2\) is the set of states whose label contains the proposition \(h_1\). Similarly, \(U_1\) is the set of states whose label contains the proposition \(p\), \(U_2\) is the set of states whose label contains the proposition \(q\), and \(U_3\) is the set of states whose label contains the proposition \(r\). Let \(\rho\) be the run of this automaton and \(\text{inf}(\rho)\) be defined as the union of all the labels of the states that are visited infinitely often. One can create a chain of \(\text{inf}\) sets such that no two consecutive sets are accepting (rejecting). In this case, one of the chains is \(\{\} \subset \{h_0\} \subset \{h_0, p, q\} \subset \{h_0, h_1, p, q\} \subset \{h_0, h_1, p, q, r\}\). The existence of a chain of length 5 demonstrates that the Rabin index of the language accepted by this automaton is 2 (there does not
exist a chain whose length is greater than 5 [CM99]). Since (7.2) describes a language with a Rabin index of 2 there does not exist an equivalent GR(1) formula.

If one considers (7.1) as the acceptance condition of this automaton then that formula translates to a Streett acceptance condition with six pairs, however, the longest chain in this automaton is of length three which matches the initial claim that (7.1) describes a GR(1) formula.
Chapter 8

Experimental Results

The hierarchy-based approach discussed in Chapter 4 has been implemented in VIS (Verification Interacting with Synthesis) [B+96]. In the remainder of this chapter, this approach is referred to as INCREMENTALGAME (IG). This approach converts the specification to an input-based game (input-based games have been discussed in Chapter 6). INCREMENTALGAME has been improved by incorporating the relation-based approach discussed in Chapter 5; this will be referred to as INCREMENTALGAME$_R$ (IG$_R$) in the remainder of this chapter. As discussed in Chapter 4, one may be able to solve a GR(1) specification through an iterative procedure. This procedure is an extension of the hierarchy-based approach and is referred to as INCREMENTALGAME$_I$ (IG$_I$) in the remainder of this chapter. Once the games are solved, one is presented with a set of strategies from which one may pick one to generate the implementation. The results being reported here use the heuristic described in [BGJ+07a].

The results of experiments conducted on a parameterized example coming from [BGJ+07a] (AMBA Bus [AL]) and [GCH11] are reported in this chapter. These specification are quite large and describe a controller that is a good stress test for the synthesis tool. Another example comes from [BGJ+07b]; it translates IBM’s description of a controller for a general buffer. Even though the specification is significantly smaller than that of an AMBA controller, its importance is due to the large number of assumptions (specification of the environment).

In the first set of results, the performance and quality of the implementations obtained by various approaches for the specification of [BGJ+07a] are compared in Table 8.2. This table includes results from ANZU [BGJ+07a] (which is a synthesis tool for GR(1) specifications) and the INCREMENTALGAME ap-
Table 8.1: Partial LTL Specification of \( n \)-Client AMBA Bus Arbiter

<table>
<thead>
<tr>
<th>Property Type</th>
<th>IG: INCREMENTALGAME</th>
<th>ANZU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. In case there are no requests, the bus is granted to MASTER(_0)</td>
<td>( G((\text{DECIDE} \land \bigwedge_i \neg \text{HB}_{i}) \rightarrow X(\text{HG}_0)) )</td>
<td>( G((\text{DECIDE} \land \forall i : \neg \text{HB}_i) \rightarrow X(\text{HG}_0)) )</td>
</tr>
<tr>
<td>2. No spurious grants except to MASTER(_0)</td>
<td>( \forall i \neq 0 . G(\neg \text{HG}_i \rightarrow (\neg \text{HG}_i \lor \text{HB}_i)) )</td>
<td>( \forall i \neq 0 . G((Q = \mathcal{I}) \land (\text{HG}_i \lor \text{HB}_i) \rightarrow X(Q = \mathcal{I})) ) ( \forall i \neq 0 . G((Q = \mathcal{I}) \land (\neg \text{HG}_i \land \neg \text{HB}_i) \rightarrow X(Q = \mathcal{I})) ) ( \forall i \neq 0 . G((Q = \mathcal{NG}) \land (\neg \text{HG}_i \land \neg \text{HB}_i) \rightarrow X(Q = \mathcal{NG})) ) ( \forall i \neq 0 . G((Q = \mathcal{NG}) \land \text{HB}_i \rightarrow X(Q = \mathcal{I})) ) ( \forall i \neq 0 . G((Q = \mathcal{NG}) \land \text{HG}_i \rightarrow \bot) )</td>
</tr>
</tbody>
</table>

\( HB = \text{HBUSREQ} \quad \text{HG} = \text{HGRANT} \)

The property \( G(\neg \text{HG}_i \rightarrow (\neg \text{HG}_i \lor \text{HB}_i)) \) is equivalent to the transition constraint \( G(\neg \text{HG}_i \rightarrow X(\neg \text{HG}_i \lor \text{HB}_i)) \).
proach with various options (IG, IG\(_R\), IG\(_I\)). The specifications used for ANZU are different from those used by INCREMENTALGAME because ANZU requires the safety components to be pre-synthesized into transition constraints, while each recurrence property must also be pre-synthesized as transition constraints and simple recurrence formulae.

The specification of the AMBA bus controller for \( n \) clients contains \( n - 1 \) properties that are \( \mathcal{R} \)-generable, but are not produced by the grammar in Definition 5.3.1. An example of such a property is shown in Table 8.1 (it has been shown in Figure 5.7 that such a property can be expressed as a transition constraint); the pre-synthesized properties that are used by the ANZU tool are also shown. The specification also contains two safety properties that are not \( \mathcal{R} \)-generable irrespective of the number of clients; these two properties are pre-synthesized by ANZU while INCREMENTALGAME and its variants do not require pre-processing of such properties. The numbers of persistence and non-persistence properties are given for both system (S) and environment (E) in Columns 2–5 of Table 8.2, while Column 6 reports the number of properties required by ANZU.

The computation times were measured on a 2.4 GHz Quad Core Pentium Duo with 4 GB of RAM. They are reported in Columns 6–9 of Table 8.2. The amount of sequential logic required by the implementations is reported in Columns 10–12, while the number of gates required by the implementation is reported in Columns 13–15. The number of gates for INCREMENTALGAME have not been collected: they are significantly larger than those of ANZU.

The table shows the advantages of the various approaches proposed in this thesis. A salient feature of both the INCREMENTALGAME\(_R\) and INCREMENTALGAME\(_I\) approaches is the significantly smaller sizes of the implementation along with much smaller times to compute them. The difference between the two approaches is that former converts the GR(1) game to a non-generalized parity game, while the latter converts the GR(1) game to a Semi-Generalized Streett(1) game and then solves it using the ITERATIVECONJGAME procedure of Section 4.3.1. One notices the significantly reduced times in computing these strategies but, contrary to expectation, the size of the implementation is not small when the input specification is quite large (as evident from tables 8.2 and 8.3). The specialized approach IG\(_I\) for GR(1) specifications is an iterative process. It converts the GR(1) specification to a special type of game (these games are called...
semi-generalized Streett(1) games and are discussed in Chapter 4) which can then be solved iteratively. This iterative process computes individual winning strategies, which need to be combined through a counter. One can combine the strategies then attempt to pick the best one or one could attempt to pick the best strategy (from the individual strategies computed by the iterative process) and then combine those individual strategies. However, in both cases selecting a strategy which utilizes minimal combinational logic is not a trivial task.

### 8.1 Importance of Retiming

In the case of the AMBA bus controller, retiming (discussed in Section 5.12) has significant impact because this controller can be implemented as a Moore machine. When a Moore implementation is not possible, the effectiveness of retiming may be less noticeable. The runtime of the algorithm is significantly reduced by retiming because the parameterized representation is also simplified: with fewer BDD variables, finding suitable variable orders becomes easier. The results of tables 8.2, 8.3, and 8.4 do not make use of conversion of general safety properties to transition constraints. Rather the automata for these properties are used directly. In the case when general safety properties are converted to transition constraints, the extraction of an optimal parameterized transition relation incurs significant penalty and is being investigated. The following example, which is a continuation of Example 5.8.4, explains the challenge encountered when general safety properties are converted to transition constraints.

**Example 8.1.1.** The steady-state solution of the variable $y'$ is defined as

$$y'_\infty = (F_2)^\neg (F_2)^\neg y' \lor (y_i \land (F_2)y').$$

When the values for $(F_2)^\neg$ and $(F_2)y'$ are plugged in the above equation, one obtains the following equation:

$$y'_\infty = (\neg r \land \neg x) \lor (r \land y) \lor (g \land y) \lor (\neg x \land y) \lor (\neg x \land \neg r' \land \neg g') \lor (r \land \neg g \land \neg r' \land \neg g').$$
All the minterms in red are don’t care conditions as the system will stay away from generating such a condition (for instance \((\neg r \land \neg x)\) will never happen as the system’s winning strategy forbids de-asserting \(r\) until a \(g\) has been witnessed; and once the environment sets \(g\) the variables \(x\) is asserted forever).

The solution of \(y'_\infty\) can be simplified when the don’t care condition are taken into account. The simplified solution is:

\[
y'_\infty = (r \land \neg g \land \neg r' \land \neg g')
\]

When using the automaton-base approach for safety properties that are not \(R\)-generable, the don’t care minimization is often significantly successful simplifying the solutions. On the other hand, when using the relation-based approach, the don’t care minimization is not as successful even when the exact same function is being simplified. This leads to a snowball effect and the size of BDDs for the solutions is quite large.

A few heuristics were tried, which resulted in significant improvements in don’t care minimization of solutions in the automata-based approach; while the same impact was not observed when all the safety properties were synthesized by the relation-based approach. Further investigation is needed to understand the root causes of this inefficiency.

In theory, the ideas discussed in development of the relation-based approach may be adopted by other automatic synthesis approaches. For instance, the safety properties of the type shown in Table 8.1 (present in both AMBA specifications) can be replaced with the respective transition constraints. This reduces the number of internal signals in the ANZU specification. Table 8.5 shows the results when ANZU synthesizes a controller with the original and the modified specification (of both AMBA specifications). This reduces the amount of sequential logic required by the implementation; however, it increases the amount of combinational logic. While the computation times have improved for for AMBA controllers with 2 masters to 12 master, they have significantly increased for the controllers with 13 and 14 masters. The drastic increase in the amount of combinational logic needs further investigation.
8.2 Optimization through ABC

The size of the handwritten implementations of AMBA shared-bus controller is compared with the results of $\text{INCREMENTALGAME}_R$ and $\text{INCREMENTALGAME}_I$ in Table 8.9. In this case, one observes that the automatic synthesis approach uses approximately 1.5 times more sequential logic. However, the amount of combinational logic starts to increase drastically as the number of clients increases. This is because it is very convenient to extend the parametrized handwritten implementation of a 2-client AMBA controller to a controller for $2 + k$ clients. On the other hand, the automatic synthesis of a $2 + k$-client AMBA controller is independent of automatic synthesis of a 2-client AMBA controller. The techniques to synthesize parametrized examples have not been explored in this thesis. Parametric synthesis has been explored in [JB12, KJB13], however, those techniques have not been applied to the AMBA specification.

One observes that the minimization through a tool like ABC (A System for Sequential Synthesis and Verification) [ABC], provides further reduction in combinational logic. The results are shown in Table 8.9. Despite the significant reduction, there is still an order of magnitude improvement required for automatic synthesis tools to start becoming competitive.
Table 8.2: Experimental Results AMBA Specification [BGJ+07a]

<table>
<thead>
<tr>
<th>Model</th>
<th>Safety</th>
<th>Parity</th>
<th>Properties</th>
<th>Time(s)</th>
<th>latches</th>
<th>gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA2</td>
<td>3 17</td>
<td>2 2</td>
<td>56</td>
<td>2.39</td>
<td>24</td>
<td>4409</td>
</tr>
<tr>
<td>AMBA3</td>
<td>4 22</td>
<td>2 3</td>
<td>68</td>
<td>44.67</td>
<td>30</td>
<td>20686</td>
</tr>
<tr>
<td>AMBA4</td>
<td>5 26</td>
<td>2 4</td>
<td>80</td>
<td>35.30</td>
<td>34</td>
<td>17501</td>
</tr>
<tr>
<td>AMBA5</td>
<td>6 31</td>
<td>2 5</td>
<td>93</td>
<td>224.06</td>
<td>39</td>
<td>48154</td>
</tr>
<tr>
<td>AMBA6</td>
<td>7 34</td>
<td>2 6</td>
<td>105</td>
<td>1011.7</td>
<td>43</td>
<td>74948</td>
</tr>
<tr>
<td>AMBA7</td>
<td>8 38</td>
<td>2 7</td>
<td>117</td>
<td>1758.5</td>
<td>48</td>
<td>88808</td>
</tr>
<tr>
<td>AMBA8</td>
<td>9 41</td>
<td>2 8</td>
<td>129</td>
<td>2034.9</td>
<td>52</td>
<td>222598</td>
</tr>
<tr>
<td>AMBA9</td>
<td>10 44</td>
<td>2 9</td>
<td>141</td>
<td>7861.2</td>
<td>57</td>
<td>175298</td>
</tr>
<tr>
<td>AMBA10</td>
<td>11 48</td>
<td>2 10</td>
<td>153</td>
<td>28319.8</td>
<td>61</td>
<td>172195</td>
</tr>
<tr>
<td>AMBA11</td>
<td>12 51</td>
<td>2 11</td>
<td>165</td>
<td>8403.3</td>
<td>65</td>
<td>179291</td>
</tr>
<tr>
<td>AMBA12</td>
<td>13 55</td>
<td>2 12</td>
<td>177</td>
<td>49138.7</td>
<td>69</td>
<td>224266</td>
</tr>
<tr>
<td>AMBA13</td>
<td>14 60</td>
<td>2 13</td>
<td>189</td>
<td>13163.4</td>
<td>73</td>
<td>239494</td>
</tr>
<tr>
<td>AMBA14</td>
<td>15 64</td>
<td>2 14</td>
<td>200</td>
<td>17104.9</td>
<td>77</td>
<td>284027</td>
</tr>
<tr>
<td>AMBA15</td>
<td>16 69</td>
<td>2 15</td>
<td>212</td>
<td>TO</td>
<td>77</td>
<td>12385</td>
</tr>
</tbody>
</table>

IG → the hierarchy based approach where the GR(1) game is converted to Streett game with 1 pair
IG_R → the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett game with 1 pair
IG_I → the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett(1) game and Theorem 4.3.3 is used to computed the winning strategy
Table 8.3: Experimental Results AMBA Specification [GCH11]

<table>
<thead>
<tr>
<th>Model</th>
<th>Safety</th>
<th>Parity</th>
<th>Properties</th>
<th>Time(s)</th>
<th>latches</th>
<th>gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E S</td>
<td>E S</td>
<td>ANZU</td>
<td>ANZU</td>
<td>IG_R</td>
<td>IG_I</td>
</tr>
<tr>
<td></td>
<td>ANZU</td>
<td>IG_R</td>
<td>IG_I</td>
<td>ANZU</td>
<td>IG_R</td>
<td>IG_I</td>
</tr>
<tr>
<td>AMBA2</td>
<td>14</td>
<td>11</td>
<td>2 2</td>
<td>41</td>
<td>1.64</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3292</td>
<td>803</td>
</tr>
<tr>
<td>AMBA3</td>
<td>16</td>
<td>15</td>
<td>2 3</td>
<td>52</td>
<td>4.91</td>
<td>1.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>29</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9708</td>
<td>1603</td>
</tr>
<tr>
<td>AMBA4</td>
<td>18</td>
<td>18</td>
<td>2 4</td>
<td>62</td>
<td>16.08</td>
<td>2.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10958</td>
<td>1837</td>
</tr>
<tr>
<td>AMBA5</td>
<td>20</td>
<td>22</td>
<td>2 5</td>
<td>73</td>
<td>71.53</td>
<td>4.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23488</td>
<td>2078</td>
</tr>
<tr>
<td>AMBA6</td>
<td>22</td>
<td>25</td>
<td>2 6</td>
<td>83</td>
<td>120.95</td>
<td>15.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>27751</td>
<td>5132</td>
</tr>
<tr>
<td>AMBA7</td>
<td>24</td>
<td>28</td>
<td>2 7</td>
<td>93</td>
<td>259.34</td>
<td>29.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>47</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46659</td>
<td>15683</td>
</tr>
<tr>
<td>AMBA8</td>
<td>26</td>
<td>31</td>
<td>2 8</td>
<td>103</td>
<td>240.74</td>
<td>44.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>43953</td>
<td>12379</td>
</tr>
<tr>
<td>AMBA9</td>
<td>28</td>
<td>35</td>
<td>2 9</td>
<td>114</td>
<td>1462.91</td>
<td>248.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55355</td>
<td>14010</td>
</tr>
<tr>
<td>AMBA10</td>
<td>30</td>
<td>38</td>
<td>2 10</td>
<td>124</td>
<td>569.68</td>
<td>281.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50317</td>
<td>26155</td>
</tr>
<tr>
<td>AMBA11</td>
<td>32</td>
<td>41</td>
<td>2 11</td>
<td>134</td>
<td>1319.33</td>
<td>606.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>88753</td>
<td>22361</td>
</tr>
<tr>
<td>AMBA12</td>
<td>34</td>
<td>44</td>
<td>2 12</td>
<td>144</td>
<td>1727.24</td>
<td>2093.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>68</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>112588</td>
<td>30742</td>
</tr>
<tr>
<td>AMBA13</td>
<td>36</td>
<td>47</td>
<td>2 13</td>
<td>154</td>
<td>3283.40</td>
<td>931.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>92216</td>
<td>31873</td>
</tr>
<tr>
<td>AMBA14</td>
<td>38</td>
<td>50</td>
<td>2 14</td>
<td>164</td>
<td>2333.88</td>
<td>10055.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>76</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110278</td>
<td>67466</td>
</tr>
<tr>
<td>AMBA15</td>
<td>40</td>
<td>53</td>
<td>2 15</td>
<td>174</td>
<td>4292.77</td>
<td>24053.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>81</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>129479</td>
<td>63244</td>
</tr>
<tr>
<td>AMBA16</td>
<td>42</td>
<td>56</td>
<td>2 16</td>
<td>184</td>
<td>5284.24</td>
<td>15701.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td>54</td>
</tr>
</tbody>
</table>

IG_{R} \rightarrow \text{the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett game with 1 pair}

IG_{I} \rightarrow \text{the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett(1) game and Theorem 4.3.3 is used to computed the winning strategy}

In this table, the retiming was not done.
Table 8.4: Experimental Results GENERAL BUFFER Specification [BGJ+07b]

<table>
<thead>
<tr>
<th>Model</th>
<th>Safety</th>
<th>Parity</th>
<th>Properties</th>
<th>Time(s)</th>
<th>Latches</th>
<th>Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E S</td>
<td>E S</td>
<td>ANZU</td>
<td>ANZU</td>
<td>IG_R</td>
<td>IG_I</td>
</tr>
<tr>
<td>GENBuf2</td>
<td>14 11</td>
<td>2 2</td>
<td>41</td>
<td>0.31</td>
<td>0.68</td>
<td>0.36</td>
</tr>
<tr>
<td>GENBuf3</td>
<td>16 15</td>
<td>2 3</td>
<td>52</td>
<td>0.53</td>
<td>3.32</td>
<td>0.51</td>
</tr>
<tr>
<td>GENBuf4</td>
<td>18 18</td>
<td>2 4</td>
<td>62</td>
<td>0.91</td>
<td>8.23</td>
<td>2.02</td>
</tr>
<tr>
<td>GENBuf5</td>
<td>20 22</td>
<td>2 5</td>
<td>73</td>
<td>1.98</td>
<td>5.19</td>
<td>2.78</td>
</tr>
<tr>
<td>GENBuf6</td>
<td>22 25</td>
<td>2 6</td>
<td>83</td>
<td>4.61</td>
<td>8.82</td>
<td>3.35</td>
</tr>
<tr>
<td>GENBuf7</td>
<td>24 28</td>
<td>2 7</td>
<td>93</td>
<td>7.16</td>
<td>38.22</td>
<td>18.91</td>
</tr>
<tr>
<td>GENBuf8</td>
<td>26 31</td>
<td>2 8</td>
<td>103</td>
<td>5.29</td>
<td>137.06</td>
<td>30.44</td>
</tr>
<tr>
<td>GENBuf9</td>
<td>28 35</td>
<td>2 9</td>
<td>114</td>
<td>5.12</td>
<td>393.53</td>
<td>69.97</td>
</tr>
<tr>
<td>GENBuf10</td>
<td>30 38</td>
<td>2 10</td>
<td>124</td>
<td>9.24</td>
<td>898.1</td>
<td>76.11</td>
</tr>
</tbody>
</table>

IG_R → the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett game with 1 pair
IG_I → the hierarchy based approach with transition constraints where the GR(1) game is converted to a Streett(1) game and Theorem 4.3.3 is used to computed the winning strategy
In this table, the retiming was not able to reduce the number of sequential elements.
Table 8.5: Comparison of Results from ANZU for AMBA Specification [BGJ+07a]

<table>
<thead>
<tr>
<th>Model</th>
<th>Time(s)</th>
<th>latches</th>
<th>gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ANZU</td>
<td>ANZU</td>
<td>ANZU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ANZU</td>
<td>ANZU</td>
</tr>
<tr>
<td>AMBA2</td>
<td>2.39</td>
<td>1.59</td>
<td>24</td>
</tr>
<tr>
<td>AMBA3</td>
<td>44.67</td>
<td>47.83</td>
<td>30</td>
</tr>
<tr>
<td>AMBA4</td>
<td>35.30</td>
<td>95.75</td>
<td>34</td>
</tr>
<tr>
<td>AMBA5</td>
<td>224.06</td>
<td>315.88</td>
<td>39</td>
</tr>
<tr>
<td>AMBA6</td>
<td>1011.7</td>
<td>974.80</td>
<td>43</td>
</tr>
<tr>
<td>AMBA7</td>
<td>1758.5</td>
<td>1284.29</td>
<td>48</td>
</tr>
<tr>
<td>AMBA8</td>
<td>2034.9</td>
<td>1240.52</td>
<td>52</td>
</tr>
<tr>
<td>AMBA9</td>
<td>7861.2</td>
<td>2877.91</td>
<td>57</td>
</tr>
<tr>
<td>AMBA10</td>
<td>28319.8</td>
<td>4439.31</td>
<td>61</td>
</tr>
<tr>
<td>AMBA11</td>
<td>8403.3</td>
<td>8458.79</td>
<td>65</td>
</tr>
<tr>
<td>AMBA12</td>
<td>49138.7</td>
<td>9136.30</td>
<td>69</td>
</tr>
<tr>
<td>AMBA13</td>
<td>13163.4</td>
<td>24045.85</td>
<td>73</td>
</tr>
<tr>
<td>AMBA14</td>
<td>17104.9</td>
<td>33020.30</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 8.6: ANZU_R represents the results of the ANZUtool when the R-generable properties of the specification (which are not transition constraints) are replaced manually with their respective transition constraints.

Table 8.7: Comparison of Results from ANZU for AMBA Specification [GCH11]

<table>
<thead>
<tr>
<th>Model</th>
<th>Time(s)</th>
<th>latches</th>
<th>gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ANZU</td>
<td>ANZU</td>
<td>ANZU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ANZU</td>
<td>ANZU</td>
</tr>
<tr>
<td>AMBA2</td>
<td>1.64</td>
<td>0.71</td>
<td>23</td>
</tr>
<tr>
<td>AMBA3</td>
<td>4.91</td>
<td>4.08</td>
<td>29</td>
</tr>
<tr>
<td>AMBA4</td>
<td>16.08</td>
<td>6.24</td>
<td>33</td>
</tr>
<tr>
<td>AMBA5</td>
<td>71.53</td>
<td>23.78</td>
<td>38</td>
</tr>
<tr>
<td>AMBA6</td>
<td>120.95</td>
<td>79.50</td>
<td>42</td>
</tr>
<tr>
<td>AMBA7</td>
<td>259.34</td>
<td>55.05</td>
<td>47</td>
</tr>
<tr>
<td>AMBA8</td>
<td>240.74</td>
<td>159.14</td>
<td>51</td>
</tr>
<tr>
<td>AMBA9</td>
<td>1462.91</td>
<td>241.08</td>
<td>56</td>
</tr>
<tr>
<td>AMBA10</td>
<td>569.68</td>
<td>315.55</td>
<td>60</td>
</tr>
<tr>
<td>AMBA11</td>
<td>1319.33</td>
<td>5210.10</td>
<td>64</td>
</tr>
<tr>
<td>AMBA12</td>
<td>1727.24</td>
<td>6567.10</td>
<td>68</td>
</tr>
<tr>
<td>AMBA13</td>
<td>3283.40</td>
<td>13858.78</td>
<td>72</td>
</tr>
<tr>
<td>AMBA14</td>
<td>2333.88</td>
<td>29995.04</td>
<td>76</td>
</tr>
<tr>
<td>AMBA15</td>
<td>4292.77</td>
<td>10508.85</td>
<td>81</td>
</tr>
<tr>
<td>AMBA16</td>
<td>5284.24</td>
<td>2445.77</td>
<td>85</td>
</tr>
</tbody>
</table>

Table 8.8: ANZU_R represents the results of the ANZUtool when the R-generable properties of the specification (which are not transition constraints) are replaced manually with their respective transition constraints.
Table 8.9: Comparison of Handwritten results of AMBA Specification [BGJ+07a] with before and after ABC Optimizations

<table>
<thead>
<tr>
<th>Model</th>
<th>BEFORE latches</th>
<th>BEFORE gates</th>
<th>AFTER latches</th>
<th>AFTER gates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H</td>
<td>IG_R</td>
<td>IG_I</td>
<td>H</td>
</tr>
<tr>
<td>AMBA2</td>
<td>8</td>
<td>15</td>
<td>13</td>
<td>426</td>
</tr>
<tr>
<td>AMBA3</td>
<td>10</td>
<td>18</td>
<td>17</td>
<td>501</td>
</tr>
<tr>
<td>AMBA4</td>
<td>11</td>
<td>20</td>
<td>18</td>
<td>562</td>
</tr>
<tr>
<td>AMBA5</td>
<td>16</td>
<td>22</td>
<td>21</td>
<td>779</td>
</tr>
<tr>
<td>AMBA6</td>
<td>16</td>
<td>23</td>
<td>22</td>
<td>800</td>
</tr>
<tr>
<td>AMBA7</td>
<td>16</td>
<td>24</td>
<td>23</td>
<td>821</td>
</tr>
<tr>
<td>AMBA8</td>
<td>16</td>
<td>26</td>
<td>24</td>
<td>842</td>
</tr>
<tr>
<td>AMBA9</td>
<td>18</td>
<td>28</td>
<td>27</td>
<td>947</td>
</tr>
<tr>
<td>AMBA10</td>
<td>19</td>
<td>29</td>
<td>28</td>
<td>1016</td>
</tr>
<tr>
<td>AMBA11</td>
<td>25</td>
<td>30</td>
<td>29</td>
<td>1430</td>
</tr>
<tr>
<td>AMBA12</td>
<td>21</td>
<td>31</td>
<td>30</td>
<td>1154</td>
</tr>
<tr>
<td>AMBA13</td>
<td>22</td>
<td>32</td>
<td>31</td>
<td>1223</td>
</tr>
<tr>
<td>AMBA14</td>
<td>23</td>
<td>33</td>
<td>32</td>
<td>1292</td>
</tr>
<tr>
<td>AMBA15</td>
<td>24</td>
<td>34</td>
<td>33</td>
<td>1361</td>
</tr>
</tbody>
</table>

Table 8.10: The AFTER results have been computed by running the **resyn2** command followed by the **fpga** command in ABC.
Chapter 9

Conclusions and Outlook

9.1 Thesis Conclusions

The purpose of this thesis research is to devise synthesis techniques that improve the quality of the implementation and close the quality gap between a handwritten implementation and an implementation obtained from an automatic synthesis approach. It has been established in this thesis that one should be very careful in converting the specification to a game, as inefficiencies and redundancies introduced during that stage are eventually inherited by the implementation. There have been moderately successful attempts at removing some of these inefficiencies and redundancies from the implementation; however, the focus of this thesis is to avoid them in the first place.

In Chapter 6 it has been established that input-based games closely resemble the actual hardware. Therefore, one can directly obtain an implementation from the winning strategies of an input-based game. It was shown that there is direct correspondence between input-based games and turn-based games: one can adapt the existing game solving algorithms for turn-based games to solve input-based games.

In Chapter 4, the procedure to convert a specification to a game in an incremental manner has been discussed. That procedure allows one to take advantage of many opportunities to simplify the game based on the Borel hierarchy of these properties. One may not be able to apply this incremental approach when the specification is a single property. However, in practice the specification is always a collection of properties of different type. An important contribution of this chapter is another efficient procedure to solve GR(1) games; this procedure leads to an efficient procedure that can solve a GR(n) game.

In Chapter 5, the relation-based synthesis framework for safety properties was discussed in detail.
The relation-based approach provides two advantages. Firstly it allows one to obtain a scalable synthesis procedure for specifications that contain a large number of safety properties. Secondly, the experimental results show that the size of the implementations is significantly reduced when the relation-based synthesis approach is used.

Finally, it was shown in Chapter 7 that there exist practically relevant reactive systems that cannot be described by a GR(1) specification in a straightforward manner. Two variations of a GR(2) specification are described based on a simplified AMBA bus specification. The experimental results in Chapter 8 show that the ideas proposed in this thesis have a significant positive impact on both the efficiency of the synthesis procedure and the quality of the implementation obtained from it.

9.2 Future Work

The results section shows that the synthesis framework based on the approaches proposed in this thesis has significantly improved the quality of results in terms of the combinational and sequential logic required by the implementation. However, it is true that the results are still not good enough when compared with handwritten implementations. It has been mentioned in the results section that don’t care minimization is not very effective when general safety properties are converted to transition constraints resulting in very poor results; a thorough investigation of the reasons behind ineffectiveness of current don’t care minimization techniques is in order.

Secondly, it has been pointed out in Section 5.4 that when one converts conjunction of several general safety properties to a transition constraints by converting each of them individually then the minimality of the augmented alphabet is not guaranteed. Further investigation is needed to improve upon the results established in this thesis.

One immediate task that needs to be addressed is the performance comparison of INCREMENTALGAME with (3.2) when the game has been obtained from a GR(1) specification. The idea is that instead of converting the game to a Semi-Generalized Streett(1) game, one could use (3.2) directly to solve the GR(1) game. However, there is no easy way to extend it to solve a game obtained from a GR(n) directly.

The effectiveness of the relation-based approach when applied to the bounded-synthesis approach of
[Ehl10] needs to be evaluated. Since bounded synthesis reduces the synthesis of an arbitrary specification to a sequence of safety games, the relation-based approach can be applied directly to this synthesis approach.

The focus of this thesis has been the synthesis of reactive systems. It is worthwhile to investigate other avenues where the ideas proposed in here can be applied. It is often argued that firmware closely resembles hardware [Gru]. It is worthwhile to investigate the automatic synthesis of firmware. It can also be argued that the quality of results may not be of extreme importance; therefore, automatic synthesis approaches may have an advantage in speed of production over handwritten implementations despite the disparity in size.

Another application of automatic synthesis is the synthesis of prototypes to be implemented in Field-Programmable Gate Arrays (FPGAs). In the case of prototype development, there is not an immediate focus on the quality of results. Therefore, the automatic synthesis approach can have an advantage over handwritten implementation because given the specification the latter requires the verification phase to establish functional correctness. However, it remains to be seen if debugging the specification itself proves to be equally challenging.
Bibliography


