CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# Symbolic solution of Emerson-Lei games

Master's thesis in Computer science and engineering

Christopher Larsson
Nils Jakobson Mo

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

# Symbolic solution of Emerson-Lei games

Christopher Larsson
Nils Jakobson Mo

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Symbolic solution of Emerson-Lei games
Christopher Larsson, Nils Jakobson Mo



Supervisor: Daniel Haussman, Department of Computer Science and Engineering
Examiner: Yehia Abd Alrahman, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Symbolic solution of Emerson-Lei games
Christopher Larsson, Nils Jakobson Mo
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Emerson-Lei games have received a lot of focus in recent research. It was only recently that the first symbolic solver algorithm for these games was introduced. As such there has not yet been any implemented game solvers for such games. In this thesis we introduce the first symbolic solver for Emerson-Lei games based on Genie and FairSyn, two game solving tools for Rabin games. We evaluate the performance of our game solver for different types of games, and compare it to FairSyn for Rabin games specifically. Further, as to motivate the practical use of such a solver, we revisit reductions from alternation free $\mu$-calculus to Büchi games and give a reduction formulated using the notation we give for Emerson-Lei games. Based on this reduction we define certificates which we used to argue the correctness of the reduction.

Keywords: Computer science, Algorithms, Logic, Emerson-Lei, Thesis, Games, Game solving, Reduction, Certificates.

# Acknowledgements

We would like to extend a huge thanks to our supervisor, Daniel Hausmann, for his extensive support throughout the project. He was a tremendous help for both the theoretical background and practical implementation. We would also like to thank Kaushik Mallik and Mateusz Rychlicki for their support regarding our implementation, which builds off of tools they took part in developing themselves.

Christopher Larsson, Nils Jakobson Mo, Gothenburg, 2024-06-12

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1

# Introduction

Many problems in theoretical computer science can be reduced to the analysis of two-player infinite-duration games. Examples of areas where such reductions can be made include model checking, logical reasoning [1], and reactive synthesis [2]. One example which we will explore in this project is model checking. In the context of games, this entails creating a game to verify if a model satisfies some specification. This can be done in a game with two players, one who tries to verify the specification and one who tries to refute it.

Since these games are connected to many different aspects of computer science, they have been subject to ongoing research. A prominent goal in this field is to find ways to analyze formal games in an efficient way. There is therefore also large interest in the development of efficient tools for game solving.

One of the more interesting types of games in recent research are Emerson-Lei games [3], first introduced in 1987 [4][5].

In this project, we implement a symbolic solver for Emerson-Lei games. Further, in part to prove the usefulness of such a solver, we revisit reductions from $\mu$-calculus models to Büchi games (for the definition of Büchi games, see Section 2.1.2), but from the perspective of Emerson-Lei games, meaning they can be solved by our implementation.

## 1.1 Formal games

A formal game in theoretical computer science can be informally defined as a game arena consisting of a graph with directed edges, and a winning condition for two or more players [6]. The game arena consists of vertices, which describe states and are marked according to players, and edges between the vertices, which represent moves in the game. These game arenas are effectively identical in structure for all games that we intend to consider in this project. What separates different types of games is the structure of their winning conditions. The winning condition is represented as one or more accepting states in the graph for a particular player, which should be visited for the winning condition of that player to hold. The winning conditions can be comprised of subconditions which have to be satisfied at different rates, some finitely often and some infinitely often.

A winning strategy is a function which tells a particular player where to go in order

to win the game, this can also be seen as a subset of the graph's edges. A path (i.e. a sequence of edges in succession) in the arena is called a play. A winning play is a play that satisfies the winning condition.

Consider the small example game in Figure 1.1, there are two players: universal and existential. The nodes from which the universal player decides the next move are marked with a square, and circles are for the existential player. The existential and universal players have different goals in this game. The existential player wants to reach the accepting state, thereby winning the game, and the universal player wants to keep the game going indefinitely. If the accepting state is never visited, the existential player loses the game since the winning condition is never fulfilled.

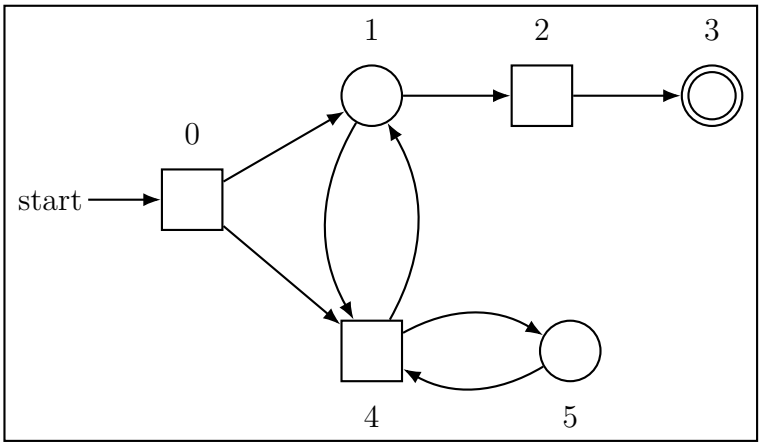


Figure 1.1: State diagram for example game.

The winning strategy for the respective players can be seen in Figure 1.2. Red edges represent the winning strategy for the universal player and green edges represent the winning strategy for the existential player. Winning strategies, specifically for reachability objectives, are subsets of edges which eventually lead the player to its goal state. The plays which satisfy the winning strategy are called winning plays, and can be seen as a path directed by the edges in the set given by the winning strategy. Note that some edges are not colored even though they are part of a winning play. This is because each player's strategy is only defined for winning nodes which the player controls.

If the universal player moves down and then right, it becomes the existential player's turn to move and it can only go back to the left. This cycle can repeat indefinitely because the universal player can, and will, continue to move back and forth from this state.

On the other hand, if the universal player were to move up, the existential player could move to the right. In this state, the universal player has no other choice than to move right and into the accepting state.

In this example, the winning condition is a reachability objective i.e. the existential player is required to reach the accepting state only once. However, as previously mentioned, games in general can have conditions which require reaching certain states infinitely often. Another type of winning condition involves avoiding certain

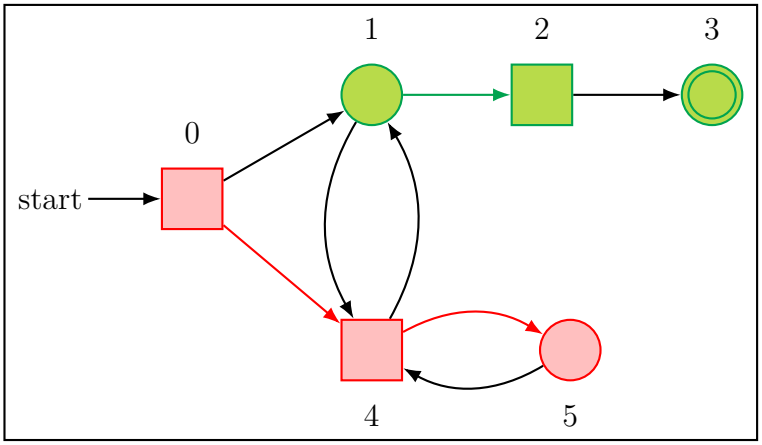


Figure 1.2: Winning strategy for universal and existential players.

states forever or from some point in time and onward. An example of this is the winning condition for the universal player.

From this example, we can see that some games are reasonably simple to solve by hand, but games in general can be arbitrarily complex and difficult to solve. Therefore, there is a demand for game solving tools. In this thesis we develop such a tool in C++. This was done using a newly presented game solving algorithm [3].

## 1.2 Reduction to games

This thesis presents a game solving implementation for Emerson-Lei games, see Section 3.1.3, and also what it can be used for. Model checking is a known problem in computer science and we will therefore present a reduction from model checking to games in Section 3.3.1. A reduction in this context is a transformation from one problem to another. This kind of reduction will make it possible for us to analyze model checking problems with our game solving method.

The reduction that we propose reduces model checking problems of the alternation-free fragment of $\mu$-calculus [7] into Büchi games, expressed using the notation we give for Emerson-Lei games.

## 1.3 Emerson-Lei games

Emerson-Lei games are a specific type of formal games in which the vertices have colors and the winning conditions are given as Boolean combinations of liveness constraints, called Emerson-Lei conditions [8]. Informally, the winning conditions are logical formulae depending on the colors of the set of visited vertices. For a complete definition of these games, see Section 2.1.

A property of Emerson-Lei games which makes them different from other kinds of games, such as Büchi, Rabin and Streett, is that two (or more) Emerson-Lei conditions can be joined together to create a new Emerson-Lei condition. This

property does not necessarily hold for other kinds of games. Emerson-Lei games are in fact a superset of the aforementioned kinds of games, as their winning conditions can be encoded using Emerson-Lei conditions.

## 1.4 Game solving and strategy extraction

Game solving and strategy extraction are two closely related topics, and as such have some overlap. Game solving involves computing the winning region i.e. the set of vertices from which a player has a winning strategy. Meanwhile, strategy extraction involves defining a function that for each vertex gives the edge which the player should move along to get closer to fulfilling the winning condition. These two concepts, winning regions and winning strategies, are naturally closely related, with game solving (i.e. extracting the winning region) often being the simpler task.

## 1.5 Genie, FairSyn, MascotSDS

Genie [9], FairSyn [10], and MascotSDS [11] are tools and libraries part of a toolchain developed to symbolically solve Rabin games [12]. Genie is a library for generating experiments on Binary Decision Diagrams (BDDs) and also supplies the virtual class for Rabin fixpoints which FairSyn and MascotSDS implement with case-specific predecessor operators. FairSyn is a tool for solving fair-adversarial and $2\frac{1}{2}$-player Rabin games [12]. MascotSDS, meanwhile, is a tool for solving abstraction-based control problems [12]. These tools also support both sequential and parallel computation of fixpoints via its BDD frontend which has built-in support for CUDD [13] (for sequential execution) and Sylvan [14] (for parallel execution). In this project we aim to modify Genie to instead support a symbolic fixpoint algorithm for Emerson-Lei games [3] and then use Fairsyn to solve Emerson-Lei games using this new backend. MascotSDS, while part of the toolchain, will not be used for our ends as it is simply out of the scope of this project.

## 1.6 Research questions

With this project we explore the following questions after implementing the game solver:

- Does our implementation produce the correct results?
- How well does our implementation perform?
- How can one efficiently evaluate (model check) formulas of the alternation-free fragment of $\mu$-calculus?

## 1.7 Thesis outline

The rest of this thesis in divided into four more chapters. Chapter 2 gives the necessary theoretical background that is needed to understand the rest of the thesis. Concepts which are used later on are also defined here. In Chapter 3, we present and explain the work we have done. Chapter 4 presents the outcome of our work, such as testing and benchmarking. Lastly, in Chapter 5, we discuss the results from Chapter 4, possible improvements to our work and limitations we have faced throughout this project. At the end of this chapter, we conclude the thesis as a whole.

# 2

# Theory

In this chapter, we present formal definitions for the concepts discussed throughout the thesis.

## 2.1 Emerson-Lei games

Emerson-Lei games, as defined in [8], are two-player infinite-duration games with winning conditions represented as Boolean formulae over the vertices of a game arena. The two players being the existential player $\exists$ (also commonly referred to as circle player, $\bigcirc$) and the universal player $\forall$ (also commonly referred to as box or square player, $\Box$). In general, the existential player wants to satisfy the winning objective and the universal player wants to prevent the winning objective from being satisfied.

A game arena $A$ is defined as a directed graph $A = (V, V_\exists, V_\forall, E)$ with $V$ being partitioned into two disjoint sets of nodes $V_\exists$ and $V_\forall$ from which opposing players can move from, in other words, $V = V_\exists \cup V_\forall$. The set of edges is defined as $E \subseteq V \times V$ and given as a function $E(v) = \{v' \in V \mid (v, v') \in E\}$ for $v \in V$. We also define a play $\pi = v_0 v_1 v_2 ...$ in the arena as a sequence of vertices $v_i \in V$ where $(v_i, v_{i+1}) \in E$ for all $i \geq 0$. The set of all plays in the arena $A$ is subsequently labeled plays$(A)$.

A game $G$ is consequently defined as $G = (A, \alpha)$, an arena $A$ and a winning objective $\alpha \subseteq$ plays$(A)$. In an Emerson-Lei game, $G = (A, \alpha_{\gamma,\varphi})$, a coloring function $\gamma : V \to 2^C$, where $C$ is a fixed set of colors, is also given. It assigns the set $\gamma(v)$ of colors to each node $v \in V$. Additionally, the objective $\alpha_{\gamma,\varphi}$ also depends on the finite Boolean formula $\varphi \in \mathbb{B}(\{\text{Inf } c\}_{c \in C})$ over atomic statements Inf $c$. This formula describes the colors which are to be seen infinitely often, and similarly, colors $c$ to be seen finitely often are described by $\neg\text{Inf } c = \text{Fin } c$. A play $\pi \in$ plays$(A)$ is then winning if and only if the set of colors visited infinitely often in $\pi$ satisfy $\varphi$. More formally, let $D_\pi$ be the set which contains all colors which are visited infinitely often in play $\pi$.

$$D_\pi = \{c \in C \mid \forall i.\, \exists j \geq i.\, c \in \gamma(v_j)\}$$

If $D_\pi \models \varphi$ then play $\pi$ is a winning play. Conversely, the game objective $\alpha_{\gamma,\varphi}$ contains all winning plays, and is formally defined in terms of $D$ as

$$\alpha_{\gamma,\varphi} = \{\pi = v_0 v_1 ... \in \text{plays}(A) \mid D_\pi \models \varphi\}$$

### 2.1.1 Strategies

A strategy $\sigma : V^* \cdot V_\exists \to V$ [3], where $V^*$ is the set of sequences of vertices, for the existential player is defined such that

$$\forall \pi \in V^*.\ \forall v \in V_\exists.\ \sigma(\pi v) \in E(v)$$

A play $\pi$ is compatible with strategy $\sigma$ if for all vertices $v_i \in \pi \cap V_\exists$ we have that $v_{i+1} = \sigma(v_0 ... v_i)$.

From this definition, the winning region $W_\exists \subseteq V$ is the set of vertices $v$ such that every play $\pi$ that starts at $v$ and is compatible with $\sigma$ is won by the existential player.

### 2.1.2 Other types of games

Formal games are differentiated by the structure of their winning objectives. Emerson-Lei games are, as mentioned in Section 1.3, a superset of some known types of games. That is, Emerson-Lei conditions directly encode combinations of other winning objectives. Some of the encodable objectives [3] are the following:

$$\begin{aligned}
&\text{Büchi}(f) && \longrightarrow \text{Inf } f \\
&\text{Generalized Büchi}(f_1, ..., f_k) && \longrightarrow \bigwedge_{1 \le i \le k} f_i \\
&\text{Rabin}(e_1, f_1, ..., e_k, f_k) && \longrightarrow \bigvee_{1 \le i \le k} (\text{Inf } e_i \wedge \text{Fin } f_i) \\
&\text{Parity}(p_1, ..., p_{2k}) && \longrightarrow \bigvee_{i \text{ even}} (\text{Inf } p_i \wedge \bigwedge_{i < j \le 2k} \text{Fin } p_j) \\
&\text{Streett}(r_1, g_1, ..., r_k, g_k) && \longrightarrow \bigwedge_{1 \le i \le k} (\text{Inf } r_i \to \text{Inf } g_i)
\end{aligned}$$

## 2.2 Zielonka trees

A Zielonka tree [15] is a tree structure for a boolean formula $\varphi$ over a set of colors $C$ defined as $\mathcal{Z}_\varphi = (T, R, l)$. Here $T$ is a set of nodes, or vertices, and $R \subseteq T \times T$ is the set of directed edges between the nodes. The labeling function $l$ is defined as $l : T \to 2^C$. It takes a node $t \in T$ as input and outputs the set of colors $C_t \in 2^C$ for the given node. This set of colors corresponds to the colors $c$ for which $Infc$ holds.

Zielonka trees are defined with a total ordering $\preceq$ which gives each node $t \in T$ an ordering depending on their placement in the tree, for example, giving the root node ordering 0 or 1.

The root node $r$ of a Zielonka tree has the label $l(r) = C$. This means that it contains every color $c \in C$. At every branching level in the tree, a minimal amount of colors is removed to alternate the satisfaction of formula $\varphi$. This means that if the label of a non-leaf node in the tree satisfies $\varphi$, its children will not. The nodes are

partitioned into sets of nodes $T = T_{\bigcirc} \cup T_{\Box}$ where nodes in $T_{\Box}$ have labels satisfying $\varphi$, and nodes in $T_{\bigcirc}$ have labels which do not satisfy the formula.

All of the child nodes with the same parent node in the tree are incomparable in the sense that they are not subsets of each other.

## 2.3 Fixpoints

Fixpoint algorithms, as we refer to them in this paper, are algorithms which aim to find a given solution to a formal game (which we previously denote $G = (A, \alpha)$) via fixpoint iteration. More precisely, the algorithm iteratively applies itself onto the input until a fixpoint is reached, that fixpoint being the solution.

The general structure of fixpoint equations [3] is given by:

$$X_i =_{\eta_i} f_i(X_1, ..., X_k)$$

for $1 \leq i \leq k$, with $\eta_i \in \{\text{GFP}, \text{LFP}\}$ and $f_i : (2^V)^k \to 2^V$. Here, GFP and LFP stand for greatest and least fixpoint and $f_i$ are monotone functions. The monotonicity of these functions means that if $A_j \subseteq B_j$ for all $j \in \{1, 2, ..., k\}$ then $f_i(A_1, ..., A_k) \subseteq f_i(B_1, ..., B_k)$.

## 2.4 Genie

Genie, the tool we use as a base for our implementation, provides most of the infrastructure we need for implementing our Emerson-Lei game solver. Most importantly it provides the infrastructure for symbolically handling sets via the use of BDDs and implements this in its fixpoint algorithm for Rabin games.

### 2.4.1 BDDs

Genie provides an abstract class for interfacing with BDDs called `UBDD` ("Universal Binary Decision Diagram"). They also provide two implementations of this class in the form of `CuddUBDD` and `SylvanUBDD` which allows for the use of the two BDD libraries CUDD [13] and Sylvan [14] respectively. These are usable for our purposes as is, and as such provide an excellent framework for the implementation of symbolic algorithms.

### 2.4.2 Fixpoint algorithm

Genie provides the abstract class for its implemented Rabin games fixpoint algorithm, which implements most of the functions leaving context dependent functions as virtual functions. Actual implementations for these functions can be found in FairSyn (and MascotSDS, although, we will not be dealing with MascotSDS).

The fixpoint algorithm implemented in Genie is in many ways similar to the one we implement, but, because of the differences between solving Rabin Games and

Emerson-Lei games, it also differs in a number of ways. One of the major differences, mentioned in Section 2.5, is the structuring of the algorithm with regards to least/greatest fixpoint. For Rabin games these fixpoint iterations follow a set pattern. For Emerson-Lei games however, the fixpoint equations depend on the winning status of nodes in the Zielonka tree and branching can happen at either a winning or losing node.

Genie's implementation also includes a number of features which are not necessarily of interest for our implementation, one of these being acceleration. Acceleration is essentially a method for reducing the amount of duplicate computations by introducing a sort of "memory" to the algorithm. Since this is strictly an optimization feature we have chosen to overlook it for this project. For similar reasons we are also focusing solely on sequential execution by only supporting the use of CUDD via their UDBB framework.

### 2.4.3 Rabin games algorithm

Genie implements an algorithm for solving Rabin Games symbolically [12], it is reproduced here in Algorithm 1.

**Algorithm 1** Symbolic Rabin games solver algorithm [12].

The symbolic fixpoint algorithm for solving Rabin games with $\mathcal{R} = \{(Q_1, R_1), ..., (Q_k, R_k)\}$ and $K = [1; k]$:

$$\nu Y_{p_0} \cdot \mu X_{p_0} \cdot \bigcup_{p_1 \in K} \nu Y_{p_1} \cdot \mu X_{p_1} \cdot \bigcup_{p_2 \in K \setminus \{p_1\}} \nu Y_{p_2} \cdot \mu X_{p_2} \cdots \bigcup_{p_k \in K \setminus \{p_1, ..., p_{k-1}\}} \nu Y_{p_k} \cdot \mu X_{p_k} \cdot \left[ \bigcup_{j=0}^{k} \mathcal{C}_{p_j} \right],$$

where

$$\mathcal{C}_{p_j} := \left( \cap_{i=0}^{j} \overline{R}_{p_i} \right) \cap \left[ \left( \mathcal{Q}_{p_j} \cap Cpre(Y_{p_j}) \right) \cup \left( Apre(Y_{p_j}, X_{p_j}) \right) \right]$$

and the definitions of *Cpre* and *Apre* are problem specific.

## 2.5 Emerson-Lei games algorithm

The symbolic algorithm for Emerson-Lei games, as proposed by Hausmann, Lehaut, and Piterman [3], is reproduced here in Algorithm 2. This algorithm forms the basis of the implemented game solver for Emerson-Lei games. Further, Hausmann, Lehaut, and Piterman proved the correctness of this algorithm in their paper.

**Theorem 1:** The existential player wins a node $v$ in an Emerson-Lei game with objective $\varphi$ if and only if $v$ is contained in the solution of the variable $X_r$ of the equation system obtained for the Zielonka tree $\mathcal{Z}_\varphi$ (where $r$ denotes the root of the tree).

**Theorem 2:** Algorithm 2 (specifically Solve($r$,[])|) computes the solution of the variable $X_r$ of the equation system obtained for the Zielonka tree $\mathcal{Z}_\varphi$.

Theorems 1 and 2 are proven to be correct in [3] and the correctness proofs are not reiterated in this thesis.

The algorithm's input parameters consist of a node in the Zielonka tree $s \in \mathcal{Z}_\varphi$ and a list $ls$ of subsets of nodes from $V$. As output, the algorithm gives a subset of nodes from $V$ which is the winning region of the given game.

**Algorithm 2** Emerson-Lei game solver algorithm (Solve(s,ls)) [3].

**if** $s \in T_\bigcirc$ **then** $X_s \leftarrow \emptyset$ **else** $X_s \leftarrow V$ ▷ Initialize variable $X_s$ for lfp/gfp
$W \leftarrow V \setminus X_s$
**while** $X_s \neq W$ **do** ▷ Compute fixpoint
  $W \leftarrow X_s$
  **if** $R(s) \neq \emptyset$ **then** ▷ Case: $s$ is not a leaf in $\mathcal{Z}_\varphi$
    **for** $t \in R(s)$ **do**
      $U \leftarrow \mathbf{Solve}(t, ls : W)$ ▷ Recursively solve for t
      **if** $s \in T_\bigcirc$ **then** $X_s \leftarrow X_s \cup U$
        **else** $X_s \leftarrow X_s \cap U$
    **end for**
  **else** ▷ Case: $s$ is a leaf in $\mathcal{Z}_\varphi$
    $Y \leftarrow \emptyset$
    **for** $t \leq s$ **do**
      $U \leftarrow \mathbf{anc}_s^t \cap \mathbf{CPre}((ls : W)(t))$ ▷ Compute one-step attraction w.r.t. $s$
      $Y \leftarrow Y \cup U$
    **end for**
    $X_s \leftarrow Y$
  **end if**
**end while**
**return** $X_s$ ▷ Return stabilized set $X_s$ as result

Note that the solver algorithm for Emerson-Lei games has a conditional for deciding if it should calculate the greatest- or least fixpoint. Meanwhile, the solver algorithm for Rabin games always calculates the fixpoints in a fixed order: greatest then least then greatest and so on. This is because a Zielonka tree for a Rabin objective only branches from the losing nodes, see Figure 2.1. This predictable behaviour arises because there are $r$ ways to make a set losing by removing a color, where $r$ is the number of remaining complete pairs in the set of colors. There is however only one way to make a node winning by removing one color.

Furthermore, in Algorithm 2, $T_\square$ represents nodes $t \in T$ which have labels $l(t)$ that satisfy the condition $\varphi$ given by $\mathcal{Z}_\varphi$. Similarly, $T_\bigcirc$ represents the nodes $t \in T$ with labels $l(t)$ that do not satisfy the condition.

The Controllable Predecessor [3] (*CPre*) function is defined as follows:

$$CPre : 2^V \to 2^V,\ CPre(X) = \{v \in V_\exists \mid E(v) \cap X \neq \emptyset\} \cup \{v \in V_\forall \mid E(v) \subseteq \emptyset\}$$

Another important function is *anc* [3], it is defined as:

$$anc_t^s = \gamma^{-1}_{\subseteq l(s)} \cap \gamma^{-1}_{\not\subseteq l(s_t)},\ s, t \in T,\ s < t$$

Here, $\gamma^{-1}_{\bowtie D} = \{v \in V \mid \gamma(v) \bowtie D\}$, for some set of colors $D \subseteq C$ and operator $\bowtie \in \{\subseteq, \not\subseteq\}$.

The initial call to Algorithm 2 is given by Solve$(r, \emptyset)$, where $r$ is the root node of the Zielonka tree.

Below we reproduce, as shown in [3], the system of fixpoint equations, $S_\varphi$, which Algorithm 2 implements. The system relates to the winning objective $\varphi$ and nodes in the Zielonka tree $s \in \mathcal{Z}_\varphi$ as follows:

$$X_s =_{\eta_s} \begin{cases} \bigcup_{t \in R(s)} X_t & R(s) \neq \emptyset,\ s \in T_\bigcirc \\ \bigcap_{t \in R(s)} X_t & R(s) \neq \emptyset,\ s \in T_\Box \\ \bigcup_{s' \leq_s} (anc_s^{s'} \cap \mathrm{CPre}(X_{s'})) & R(s) = \emptyset \end{cases}$$

This definition is for $s \in T$ and $X_t$ refers to $X_i$ where $i$ is the index of $t$ in the total ordering of the nodes in the Zielonka tree. LFP is used if $t \in T_\bigcirc$ and GFP is used if $t \in T_\Box$.

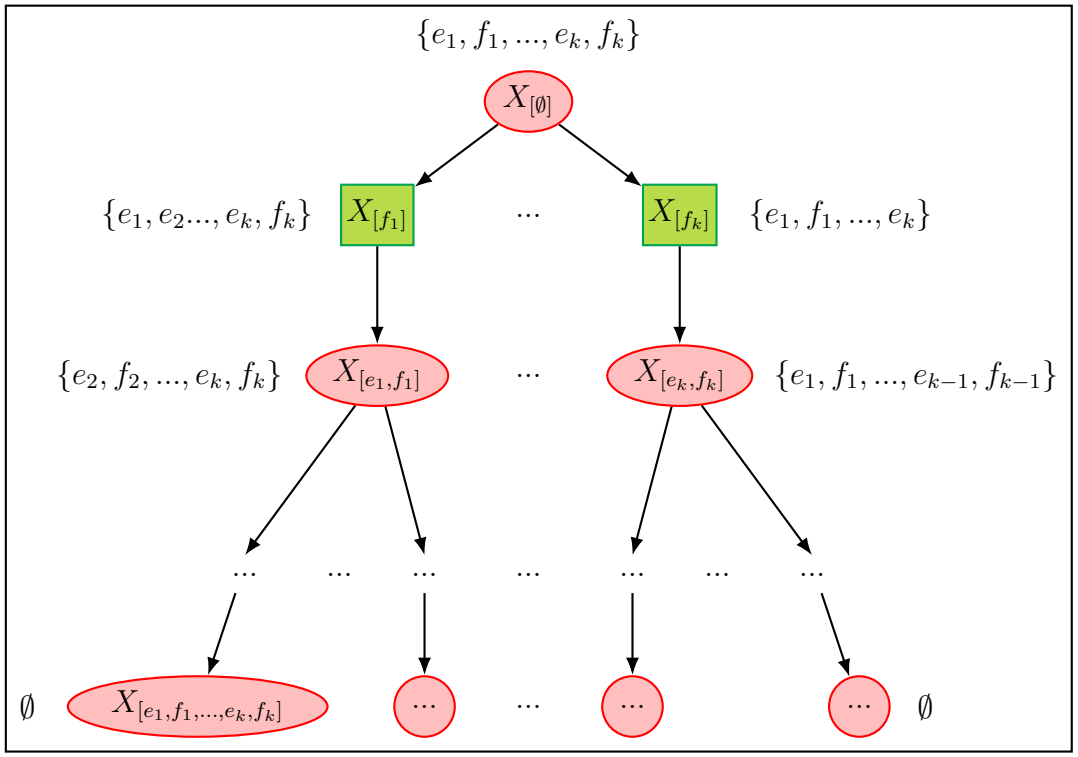


Figure 2.1: Zielonka tree of a Rabin objective $\bigvee_{1 \leq i \leq k} (\mathrm{Inf}\ e_i \wedge \mathrm{Fin}\ f_i)$.

## 2.6 The alternation-free µ-calculus

In this section we give the necessary definitions for formulating the reduction from model checking to Büchi games, which we present in Section 3.3.

### 2.6.1 Model checking

Model checking is a concept in computer science that involves checking if a model $\mathcal{M}$ satisfies some specification $\varphi$. This can also be written as $M \models \varphi$.

Given a model $\mathcal{M} = (S, T, \mathcal{V})$ where $S$ is a set of states and $T \subseteq S \times S$ is a set of state transitions, model checking wants to answer if specification $\varphi$ is fulfilled in the model or not. The model also includes a valuation function $\mathcal{V}$ to valuate which atomic statements are satisfied in each state.

### 2.6.2 Reduction

A reduction in this context of model checking and games is a translation from one to the other. From a fixed set of rules, a reduction converts a given model into a game which deconstructs the model into smaller pieces. The correctness of a reduction is the requirement that an instance of a problem has a positive answer if and only if a reduced instance of the problem also has a positive answer.

### 2.6.3 Syntax and semantics

We formulate a syntax for the alternation-free fragment of $\mu$-calculus [7], where $Props$ is a set of propositional variables and $Vars$ is a set of fixpoint variables. This grammar and syntax is defined in a standard way that, for the most part, resembles the formulation Leucker gives in his paper [1].

$$\varphi ::= \top \mid \bot \mid P \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Diamond\varphi \mid \Box\varphi \mid \nu X.\varphi \mid \mu X.\varphi \mid X$$

Where $P \in Props$ and $X \in Vars$. We also define the function $Sub(\varphi)$ in a standard way:

$$\begin{aligned}
Sub(\top) &= \{\top\} \\
Sub(\bot) &= \{\bot\} \\
Sub(P) &= \{P\} \\
Sub(X) &= \{X\} \\
Sub(\varphi_1 \wedge \varphi_2) &= \{\varphi_1 \wedge \varphi_2\} \cup Sub(\varphi_1) \cup Sub(\varphi_2) \\
Sub(\varphi_1 \vee \varphi_2) &= \{\varphi_1 \vee \varphi_2\} \cup Sub(\varphi_1) \cup Sub(\varphi_2) \\
Sub(\Diamond\varphi) &= \{\Diamond\varphi\} \cup Sub(\varphi) \\
Sub(\Box\varphi) &= \{\Box\varphi\} \cup Sub(\varphi) \\
Sub(\nu X.\varphi) &= \{\nu X.\varphi\} \cup Sub(\varphi) \\
Sub(\mu X.\varphi) &= \{\mu X.\varphi\} \cup Sub(\varphi)
\end{aligned}$$

The $Sub$ function recursively deconstructs different formulas into their constituent parts, down to and including their atom statements.

Following the syntax of our fragment of $\mu$-calculus, we defined the semantics of

satisfaction for model $\mathcal{M} = (S, T, \mathcal{V})$ at state $s \in S$ as follows.

$$
\begin{aligned}
&\mathcal{M}, s \models_{\mathcal{V}} \top \\
&\mathcal{M}, s \not\models_{\mathcal{V}} \bot \\
&\mathcal{M}, s \models_{\mathcal{V}} P && \text{iff } s \in \mathcal{V}(P) \\
&\mathcal{M}, s \models_{\mathcal{V}} X && \text{iff } s \in \mathcal{V}(X) \\
&\mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \wedge \varphi_2 && \text{iff } \mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \text{ and } \mathcal{M}, s \models_{\mathcal{V}} \varphi_2 \\
&\mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \vee \varphi_2 && \text{iff } \mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \text{ or } \mathcal{M}, s \models_{\mathcal{V}} \varphi_2 \\
&\mathcal{M}, s \models_{\mathcal{V}} \Diamond\varphi && \text{iff } \exists t \in E_M(s) : \mathcal{M}, t \models_{\mathcal{V}} \varphi \\
&\mathcal{M}, s \models_{\mathcal{V}} \Box\varphi && \text{iff } \forall t \in E_M(s) : \mathcal{M}, t \models_{\mathcal{V}} \varphi \\
&\mathcal{M}, s \models_{\mathcal{V}} \nu X.\varphi && \text{iff } \exists U \subseteq 2^S,\ s \in U \text{ and } \forall t \in U : \mathcal{M}, t \models_{\mathcal{V}[X/U]} \varphi \\
&\mathcal{M}, s \models_{\mathcal{V}} \mu X.\varphi && \text{iff } \forall U \subseteq 2^S,\ \text{if } s \notin U \text{ then } \exists t \in S : t \notin U \text{ and } \mathcal{M}, t \models_{\mathcal{V}[X/U]} \varphi
\end{aligned}
$$

Here, $\mathcal{V} : \mathit{Props} \cup \mathit{Vars} \to 2^S$ is the valuation which maps variables $X$ to subsets of $S$, $\mathcal{V}(X) \subseteq S$. The valuation map $\mathcal{V}$ has a substitution syntax $\mathcal{V}[X/U]$ which replaces all occurrences of $X$ with $U$ at valuation of a formula.

The function $\Theta(X)$ takes a fixpoint variable $X$ as input and outputs the formula which the variable is bound to. This definition requires *cleanness*, which means that $X$ can only be bound to at most one formula. The formulation we use also requires *alternation-freeness*, which disallows greatest and least fixpoints to be nested.

Syntax $\mathcal{M}, s \models_{\mathcal{V}} \varphi$ means that model $\mathcal{M}$ satisfies $\varphi$ at state $s$ under the valuation $\mathcal{V}$.

# 3

# Methods

This chapter describes the reduction we proposed and implementation details of our game solver. It is based on the game solving tool FairSyn [10] which uses Genie [9] as a backend.

The main programming languages we use are C++, Python and Bash. These three languages work well in tandem. C++ is used as the main driver and performs all computationally demanding tasks. Python is used for visualisation of graphs and statistical presentation. Bash is used as a scripting language for execution and testing.

Make and CMake are used to simplify the compilation and execution of our various programs.

## 3.1 Genie-EL

This section presents the changes we have made in our fork of Genie, which we have decided to call Genie-EL.

### 3.1.1 Zielonka tree

The Zielonka tree is a structure closely related to the solving of Emerson-Lei games. This section describes the behaviour, methods and structures developed and used during the implementation process.

#### 3.1.1.1 Data structures

To maintain a good structure for the implementation of the Zielonka tree, we created a class `ZielonkaTree` which in turn contains a member variable named `root` of type `ZielonkaNode*`. We implemented it this way to encapsulate data easily and with efficient use of space.

The `ZielonkaNode` struct is what creates the tree structure of the `ZielonkaTree` class. Each instance of this structure contains an array `children` of directly linked child nodes. This array is empty at initialisation but is then populated during the execution of the method `generate()`, belonging to the `ZielonkaTree` class.

Along with the array of the child nodes is an array containing the set differences of the current node and its directly linked children. This is used for the fixpoint computations in Listing 4.

The `ZielonkaNode` struct, see Listing 1, also contains the following: a pointer to its parent node, the ordering of its parent node, a label of colours, a field `level` which represents the depth of the node in the tree, the order of the node in the tree and the boolean value `winning` which is true if the node satisfies the given winning condition and false otherwise.

```
struct ZielonkaNode {
    std::vector<ZielonkaNode*> children;
    std::vector<std::vector<bool>> child_differences;
    ZielonkaNode *parent;
    size_t parent_order;
    std::vector<bool> label;
    size_t level;
    size_t order;
    bool winning;
};
```

Listing 1: Definition of the ZielonkaNode data structure.

#### 3.1.1.2 ZielonkaTree class

The Zielonka tree is implemented as its own class, see Listing 2, instantiated with an integer argument representing how many different colours are given in the formula for the winning condition. It has two member variables `root` and `phi`. The `root` variable, of type `ZielonkaNode*`, contains the highest level node in the tree structure and `phi` is the Emerson-Lei condition.

The constructor function has the behaviour shown in Listing 3. It takes two arguments, `colors` and `conditionFile` as input. Parameter `colors` is a positive integer which represents the number of colors in the Boolean formula `phi`. Parameter `conditionFile` holds the path to the file from which `phi` is generated. From `colors`, the member variable `root` is instantiated with empty arrays `children` and `child_differences`. The parent node is set to be `nullptr`. The order of the root node is 1, and therefore the parent of this node has order 0. The array `label` of colors is declared as an array with length corresponding to the constructor argument `colors` and all elements are set to true. Since a Zielonka tree's root node contains all colors, the `winning` field is set to true or false depending on the `evaluate_phi` function, see Section 3.1.2.

```
class ZielonkaTree {
private:
    // Private Variables
    ZielonkaNode *root;
    // Emerson-Lei condition in tokenized postfix format
    std::vector<std::string> phi;

    // Private methods
    void generate();
    void generate_parity();
    void generate_phi(const char*);
    bool evaluate_phi(std::vector<bool>);
    void displayZielonkaTree();
    void graphZielonkaTree();

public:
    ZielonkaTree(const char*, size_t);
    ~ZielonkaTree() {};
    ZielonkaNode* get_root();
};
```

Listing 2: Definition of the ZielonkaTree class.

```
ZielonkaTree::ZielonkaTree(const char* conditionFile,
                           size_t colors) {
    generate_phi(conditionFile);
    std::vector<bool> label(colors, true);
    root = new ZielonkaNode {
        .children  = {},
        .child_differences = {},
        .parent = nullptr,
        .parent_order = 0,
        .label = label,
        .level = 1,
        .order = 1,
        .winning = evaluate_phi(label)
    };
    generate();
    graphZielonkaTree();
    std::cout << "leaves: "<< leaves << '\n';
    std::cout << "nodes: " << total_nodes  << '\n';
    displayZielonkaTree();
    graphZielonkaTree();
}
```

Listing 3: The constructor function of the ZielonkaTree class.

The method `generate()`, see Listing 6 in Appendix A, is what populates the whole tree. The way it works is the following: keep a queue initialized with only the root node and create a powerset from all colors. Sort the powerset in descending order in terms of how many elements in the array are true. Keep a local counter of the current node's order and a nested array of colors which have been used in connection to the same parent node.

While the queue of nodes is not empty, pop the next node off the queue and mark it as the current node, $z$. Check for every set $c$ in the powerset if $c$ is a strict subset of $label(z)$. If it is, check that $c$ is not a subset of any of the current children of $z$. If it is not, check that $c$ has the opposite winning status as $z$. So if $label(z)$ satisfies the condition, then $c$ should not.

If all of these conditions are met, add a `ZielonkaNode*` to the array of children of $z$ and initialize it accordingly. Finally, add the set difference of $c$ and $label(z)$ to the `child_differences` array of $z$ and push the new node onto the queue.

When the queue is empty, all of the branches of the tree have been generated and the Zielonka tree is ready to use.

In the case that the Zielonka tree is supposed to be generated from a Parity objective, another function called `generate_parity` can be used. This function uses the fact that Zielonka trees from Parity objectives always only have one way to convert

a losing state to a winning state and vice versa. The `generate_parity` function therefore just iterates over the set of colors and removed one in each step. This way of generation gives a drastic speedup to the generation and runs in linear time. It is however just a prototype and has not gone through thorough testing.

### 3.1.2 Condition evaluation

To perform Zielonka tree generation, a way to evaluate if a subset of colors satisfies the given winning objective is required. In this section, we present the methods we have used to achieve this condition evaluation.

#### 3.1.2.1 Colors

In this implementation, the concept of a set of colors is represented as an array of boolean values. For example, assume that the following condition with three colors is given:

$$\varphi = \text{Fin } a \wedge (\text{Inf } b \vee \text{Inf } c)$$

In our parsing we have the following input grammar, with colors `c`:

```
a,b ::= c | !c | Inf c | Fin c | a|b | a&b
```

and therefore the condition would be given as:

```
!0 & (1 | 2)
```

or alternatively as:

```
Fin 0 & (Inf 1 | Inf 2)
```

We get 3 distinct colors $a$, $b$, $c$, which get their own index in an array of boolean values, index 0 represents $a$, index 1 represents $b$ and index 2 represents $c$. The element at index $i$ in the array represents the presence of color $c_i$ from the input formula in a node's label, indicating that it satisfies Inf $c_i$. At a node which has label $\{a, c\}$, the array would therefore look like the following:

```
{true, false, true}
```

This set would not satisfy the condition because color $a$ is supposed to only appear finitely often.

#### 3.1.2.2 Evaluation function

The first evaluation function we implemented used a Python script which read the condition from a file and then created a C++ header file with a hard-coded inline function. This function would then be used to evaluate the condition from a specified array of Boolean values. We identified a considerable practical downside to this: since the file with the generated evalution function was imported by Genie, we

would have to re-run the Python script then re-build Genie and FairSyn every time we wished to run a test using a new condition.

Because of this, we opted to implement a replacement, written entirely in C++, for the Python script and its generated function. This instead lets us test different conditions by simply changing the input to FairSyn. This new implementation takes the same kind of input as the previous solution, tokenizes it and then translates it into postfix notation, which is how it is stored. This postfix expression is stored in the `ZielonkaTree` structure and is, similarly to our previous implementation, evaluated by calling an evaluation function which takes a vector of Booleans as input. This way of implementing the evaluation is linear in time complexity, which makes it scale well with respect to input size.

#### 3.1.2.3 Visualization

One way to test that the Zielonka tree generation works as intended is to graph out the tree visually. For that reason, we created a simple terminal based tree graphing function. As an example, given the Rabin objective with 2 Rabin pairs:

$$\varphi = (\text{Fin } a \wedge \text{Inf } b) \vee (\text{Fin } c \wedge \text{Inf } d)$$

the graphing function outputs an ASCII Zielonka tree representation, as seen in Figure 3.1.

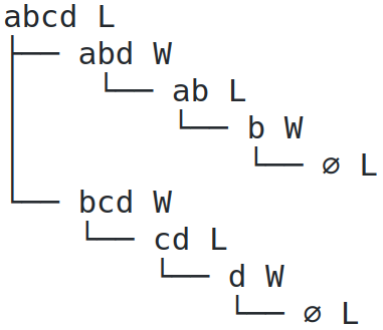


Figure 3.1: ASCII representation of Zielonka tree from Rabin objective with 2 Rabin pairs.

In this representation, the first string of characters represents a node's label of colors $c$ which satisfy Inf $c$. If $\varnothing$ is shown, then there are no colors satisfying Inf $c$. `L` (losing) and `W` (winning) represent the winning status of the node, that is, if the node's label satisfies the condition or not.

Two slightly larger examples are included in Figures A.2 and A.3.

### 3.1.3 Algorithm implementation

The extension that we have made to Genie includes the Zielonka tree functionality, presented in Section 3.1.1, and the solving algorithm for Emerson-Lei games, see

Algortihm 2. The algorithm implementation is included in our fork of Genie, which makes it easily accessible and keeps the previous functionality of Genie intact.

Before this function is called, a Zielonka tree is generated from the given objective and set of colours. The `EmersonLei` function then takes a node from the generated Zielonka tree as input when called. In the initial call, the node argument is the root of the Zielonka tree.

```
UBDD EmersonLei(ZielonkaNode *t,
                UBDD term) {
    UBDD right = term;
    UBDD U, Y, YY;

    if (t->winning) {
        Y = base_.zero();
        YY = base_.one();
    } else {
        Y = base_.one();
        YY = base_.zero();
    }
    UBDD term1 = base_.one();
    for (size_t i = 0; i < t->label.size(); ++i){
        if (!(t->label[i]))
            term1 &= (color_UBDDs[color_UBDDs.size()/2 + i]);
    }
    for (int j = 0;
         Y.existAbstract(CubeNotState()) !=
         YY.existAbstract(CubeNotState());
         j++) {
        Y = YY;
        if (t->children.empty())
            YY = right | term1 & cpre(Y);
        else {
            if (t->winning)
                YY = base_.one();
            else
                YY = base_.zero();
            for (auto s : t->children) {
                UBDD term2 = base_.zero();
                std::vector<bool> diffst = ELHelpers::label_difference(
                                                t->label,
                                                s->label
                                            );
                for (size_t i = 0; i < diffst.size(); ++i){
                    if (diffst[i])
                        term2 |= color_UBDDs[i];
                }
                UBDD term3;
                term3 = right | (term1 & term2 & cpre(Y));
                U = EmersonLei(s, term3);
                if (t->winning) {
                    YY &= U;
                } else {
                    YY |= U;
                }
            }
        }
    }
    return YY;
}
```

Listing 4: Implementation of the Emerson-Lei game solving algorithm presented in Algorithm 2.

Listing 4 shows the final version of the implementation. The Emerson-Lei function is a method of the `genie::BaseFixpoint<UBDD>` class. It takes as input a `ZielonkaNode*` and a `UBDD`.

The function starts off by defining fixpoint variables `U`, `Y`, `YY` and preparing `Y` and `YY` for either the greatest or least fixpoint iteration. After this is done, a loop over the colors in the label of Zielonka node `t` is done. For every color which is marked 0, the `term1` accumulates the negated version of that color's BDD. In the end, `term1` contains the intersection of the color BDDs for the colors which are not contained in the label of `t`.

After this process, the fixpoint computations are carried out according to if the Zielonka node `t` is winning or not. The looping behaviour continues until the greatest or least fixpoint is obtained. In the beginning of each iteration, `Y` takes the value of `YY` to keep track of any changes which might occur during the current iteration.

During iteration, if the Zielonka node `t` is a leaf node, `YY` receives the value of a union between the accumulative BDD `term`, which is taken in as a parameter, and the negated color intersection `term1`. This union is then intersected with CPre(`Y`).

If `t` is not a leaf node, for every direct child node $s \in R(t)$, the BDD `term2` is intersected by the colors in $t \setminus s$. BDD `term3` is then defined as the union of `term` and the intersection between `term1`, `term2` and CPre(`Y`). In the recursive solution call, `term3` is passed as the accumulative argument.

If `t` is winning, `YY` is intersected with the recursive solution and otherwise the intersection is relaced with a union.

After all of the iteration is done, `YY` is returned, describing the stabilized winning region of the game.

Important to note is that our implementation does not correlate fully to Algorithm 2. Rather than computing CPre only in leaf nodes, we compute CPre in all nodes which are not leaves. As a result of this, there are in essence two main differences between our implementation and the algorithm as it is given in Section 2.5. The first difference is that our implementation avoids having to use $anc_s^t$. The second difference is that when leaf nodes are reached the algorithm simply continues with `YY` as the accumulated result. Another difference is that we precompute the inverses of the color BDDs and put them at the end of the `color_UBDDs` vector. This is to reduce the amount of computations needed during the runtime of the solution.

## 3.2 FairSyn-EL

FairSyn-EL, our fork FairSyn, is the tool that we decided to use as a front-end for the game solver. Since FairSyn is already capable of solving Rabin games with some fairness constraints, we did not need to modify a lot of the code.

The first thing we did was to add a separate executable for generating a Zielonka tree and calling the `EmersonLei` function. This new executable, called *main_fairsyn_EL*, is generated from the source file *main_fairsyn_EL.cc*. To be clear, the file called

*main_fairsyn_EL.cc* is very similar to the original source file *main_fairsyn.cc*, but only includes code necessary for solution of Emerson-Lei games. In this file, an array of color BDDs is also introduced and passed to the `Fixpoint` class defined as a subclass of Genie's `BaseFixpoint` class.

Also introduced in this file is a configuration parser which not only parses nodes and transitions, but also reads the label of each node.

Parsing is done by reading the universal nodes, existential nodes, colors and lastly transitions. Each line in the arena configuration file is terminated with $-1$ and each set, such as the colors or transitions, is terminated with $-2$.

Listing 5 shows what an arena configuration file might look like, annotated with comments. Figure 3.2 shows the arena specified in the configuration file.

```
// Arena configuration

1 2 -1   // V∀ = {v1, v2}

0 -1     // V∃ = {v0}

1 -1     // γ⁻¹(c0) = {v1}
2 -1     // γ⁻¹(c1) = {v2}
-2

1 2 -1   // E(v0) = {v1, v2}
0 -1     // E(v1) = {v0}
0 -1     // E(v2) = {v0}
-2
```

Listing 5: An example of an arena configuration file.

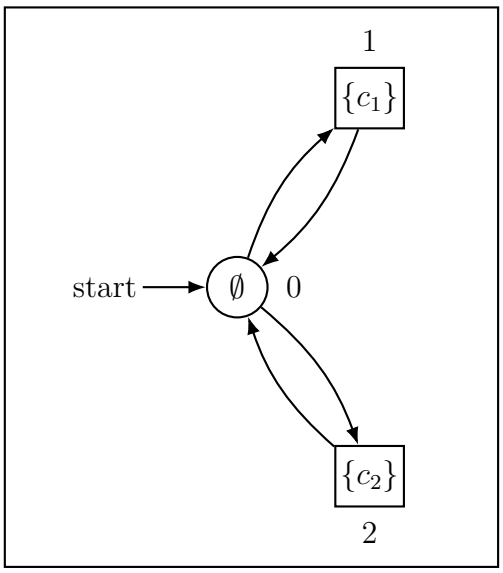


Figure 3.2: An example of a game arena.

### 3.2.1 Usage

Since the tools which form the basis of our implementation, Genie and FairSyn, are open source we decided to make our forks open source as well. In this section we describe how to set up and run the game solver.

Firstly, clone the Genie-EL [16] and FairSyn-EL [17] repositories from GitHub.

```
git clone git@github.com:chrilars/genie-EL.git
git clone git@github.com:chrilars/fairsyn-EL.git
```

After this, follow the installation guides included in the `README.md` files for the

respective repositories and verify that the projects compile without any errors. Note that Fairsyn-EL depends on Genie-EL, so Genie-EL needs to be compiled first.

When this is done, the project is ready to use. The only thing left to do is to create files specifying objective and arena. To run the solver, execute the following commands from the `fairsyn-EL` root path.

```
cd build/bin/
./main_fairsyn_EL \
    <info-file-path> <winning-region-file-path> \
    <arena-file-path> <objective-file-path>
```

When the program has finished running, the `<info-file>` will contain the following information:

```
afp ofp wrfp n t rt mu
```

These fields are in order and are defined as follows:

- `afp`: Path to given arena configuration file
- `ofp`: Path to file with winning objective
- `wrfp`: Path to file which stores the winning region
- `n`: Number of nodes in the arena
- `t`: Number of transitions in the arena
- `rt`: Runtime of the game solution
- `mu`: Maximum memory usage during the game solution

The file specified as `<winning-region-file>` stores the winning region. The non-negative numbers represent the nodes which are in the winning region and the negatives ones represent the rest of the nodes in the arena.

## 3.3 Reduction from μ-calculus to Büchi games

Here we present our reduction from $\mu$-calculus models to Büchi games, the definition of certificates, and a correctness argument for the reduction building on this notion of a certificate.

### 3.3.1 Reduction from µ-calculus to games

We define the starting node in the game as $(s_0, \varphi_s) \in S \times Sub(\varphi)$ where $s_0$ is the starting node in the model and $\varphi_s$ is the specification in its entirety. We then define a function $E_G(s, \varphi)$ which defines the successors of $(s, \varphi)$ in the game arena. The rest of the game nodes are then defined recursively by applying $E_G(s, \varphi)$ to the starting node, then its children and their children and so on, until all branches have either reached a *leaf* node (per the rules for $\varphi = \top \mid \bot \mid P$) or looped back to an already defined game node (per the rule for $\varphi = X$).

We also define the following sets: $V_\exists$, which contains all nodes belonging to the existential player, and $V_\forall$, which contains all nodes belonging to the universal player. They are defined as follows (where $\varphi$, $\varphi_1$, and $\varphi_2$ are all elements of $Sub(\varphi_s)$):

$$V_\exists = \{(s, \varphi) \mid \varphi \in \{\bot, \varphi_1 \vee \varphi_2, \Diamond\psi\}\} \cup \{(s, P) \mid s \in \mathcal{V}(P)\}$$
$$V_\forall = \{(s, \varphi) \mid \varphi \in \{\top, \varphi_1 \wedge \varphi_2, \Box\psi, \nu X.\varphi, \mu X.\varphi, X\}\} \cup \{(s, P) \mid s \notin \mathcal{V}(P)\}$$

The definition of $E_G(s)$, a function which takes a state and subformula and outputs successor nodes in the reduction, is as follows (where $E_M(s)$ gives the successors of state $s$ in the model):

$$\begin{aligned}
E_G((s, \top)) &= \{(s, \top)\} \\
E_G((s, \bot)) &= \{(s, \bot)\} \\
E_G((s, P)) &= \{(s, P)\} \\
E_G((s, \varphi_1 \wedge \varphi_2)) &= \{(s, \varphi_i) \mid i \in \{1, 2\}\} \\
E_G((s, \varphi_1 \vee \varphi_2)) &= \{(s, \varphi_i) \mid i \in \{1, 2\}\} \\
E_G((s, \Diamond\varphi)) &= \{(t, \varphi) \mid t \in E_M(s)\} \\
E_G((s, \Box\varphi)) &= \{(t, \varphi) \mid t \in E_M(s)\} \text{ if } E_M(s) \neq \emptyset \\
E_G((s, \Box\varphi)) &= \{(s, \top)\} \text{ if } E_M(s) = \emptyset \\
E_G((s, \nu X.\varphi)) &= \{(s, \varphi)\} \\
E_G((s, \mu X.\varphi)) &= \{(s, \varphi)\} \\
E_G((s, X)) &= \{(s, \Theta(X))\}
\end{aligned}$$

For the alternation-free fragment of $\mu$-calculus we define the winning condition as a Büchi condition, Inf $p$ ($p$ can be seen as the "winning" color), such that for all nodes $n$:

$$l(n) = \{p\} \text{ if } n \in \{(s, \varphi) \mid \varphi \in \{\top, \nu X.\psi\}\} \cup \{(s, P) \mid s \in V(P)\}$$
$$l(n) = \emptyset \quad \text{otherwise}$$

The reduction we are proposing could visually be represented by Figures 3.3 and 3.4. There, an example model checking graph is given along with a formula $\varphi$. This formula is formulated in both Computational tree logic (CTL) and $\mu$-calculus, and the semantics of $\varphi$ is "there exists a path in the graph for which $p$ eventually holds".

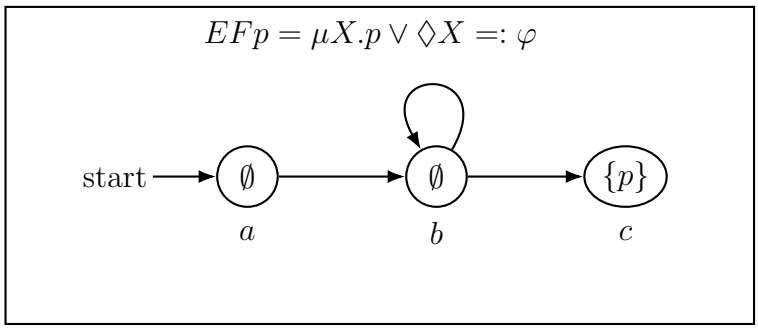


Figure 3.3: Example model and specification.

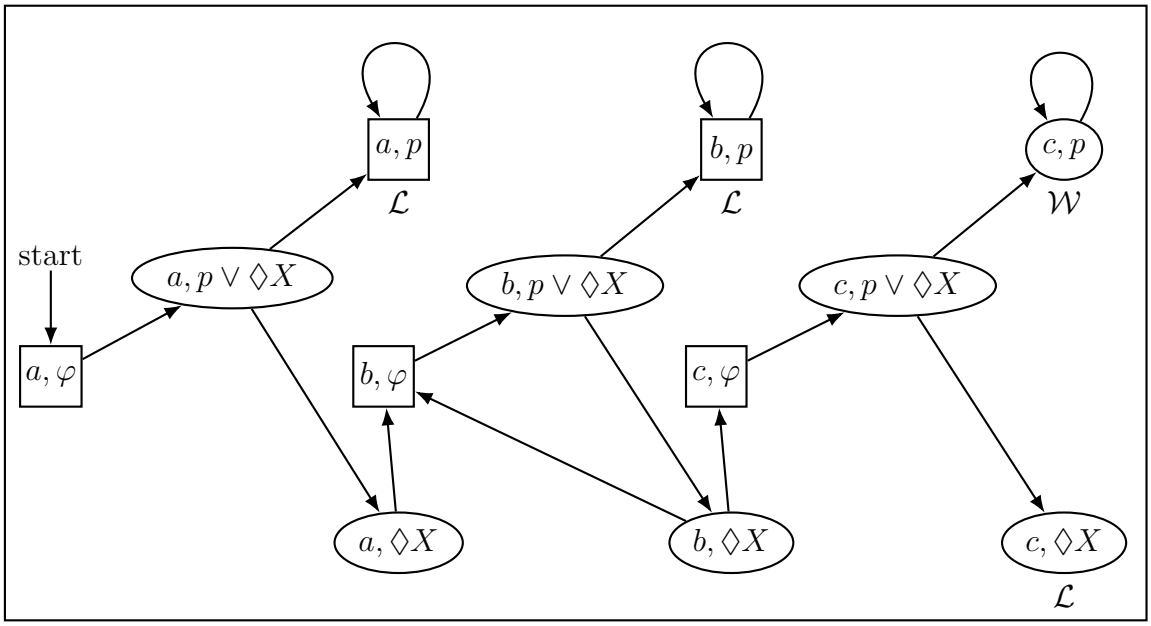


Figure 3.4: Reduction of an example game represented by a graph. Here, some nodes are left out because of redundancies.

### 3.3.2 Certificates of model satisfaction

To further argue for the correlation between reduction games solutions and model satisfiability, we define a certificate $cert_\sigma : S \times Sub(\varphi) \to 2^{S \times Sub(\varphi)}$ for a strategy $\sigma : S \times Sub(\varphi) \to S \times Sub(\varphi)$.

Given a specification $\varphi$ and a model $\mathcal{M} = (S, T, \mathcal{V})$, where $S$ is a set of states and $T \subseteq S \times S$ are edges, we construct a reduction arena $A = (V, V_\exists, V_\forall, E)$ where $V : S \times Sub(\varphi)$, $V_\exists \cup V_\forall$ and $E \subseteq V \times V$. The construction of $A$ is according to the reduction rules and winning condition described in Section 3.3.1.

Application of strategy $\sigma$ on every node $v \in V$ yields the next node in the path $\pi$ which satisfies the condition $\varphi$. Similarly, application of certification $cert_\sigma$ on a node in $A$ outputs a set of nodes, excluding nodes and edges not given by $\sigma$.

We can calculate the certificate $cert_\sigma$, $W$ being the winning region of the reduced game, $(V_W, E_W)$ being the nodes and edges in the winning region, respectively, as

follows:

$$
\begin{aligned}
V_W &= V_{W\forall} \cup V_{W\exists} \\
W &= (V_W, E_W) \\
E_\sigma &= \bigcup_{v \in V_W} (v, \sigma(v)) \\
E_{W\Box} &= \{e = (v_1, v_2) \mid e \in E_W,\ v_1 \in V_{W\forall}\} \\
cert_\sigma &= (V_W, E_{W\forall} \cup E_\sigma)
\end{aligned}
$$

Note that this resulting graph, $cert_\sigma$, is a sub-graph of the winning region $W$ such that all possible plays in $cert_\sigma$ are plays which satisfy the winning strategy $\sigma$. However, this only holds as long as the strategy $\sigma$ is memory-less.

The reason we introduce this notion of a *certificate*, and why we named it as such, is because the certificate shows *why* the given specification holds for the given model. It does this by giving all the nodes which, through their inclusion, says that the subformula of the node can always be fulfilled from the state in the node. Consequently the set of all these nodes, as given by $cert_\sigma$, form a complete proof for why $\varphi$ holds. It is important to note, though, that this proof is not necessarily unique. If there exists another strategy $\sigma_2 \neq \sigma$ then the proof constructed by $cert_{\sigma_2}$ will differ from the proof given by $cert_\sigma$ but both will be valid proofs.

Why this proof might be useful from a practical viewpoint is because it can be used to manually verify that the specification holds. Computing a winning region in the reduced game shows that the specification holds, but does not show why. A strategy shows how the existential player wins in the reduced game, but by itself it does not show why the specification holds. The certificate uses the strategy in the reduced game to show why the specification holds for the model.

### 3.3.3 Correctness of the reduction

We will here argue for why the reduction we present in Section 3.3 is correct, that is ($V_W$ being the winning region in the game):

$$\mathcal{M}, s \models \varphi \text{ iff } (s, \varphi) \in V_W$$

The argument will be structured on a per-rule basis, where we will first argue that for each rule we have defined for the reduction, we create a game node $(s, \varphi)$ from which existential player has a winning strategy iff the formula $\varphi$ is satisfied in state $s$. Second, we will argue that if the existential player wins the reduced game then the formula is satisfied.

The principle behind the construction of our reduction is that the reduced game shows that a specification holds for a given model by, for every game node, making claims about parts of the specification for a specific state in the model. Consequently, the existential player winning the game (should) correspond to a play that at some point reaches a node $(s, \top)$ or $(s, P)$ where $s \in \mathcal{V}$. If such a node is never reached,

the play should instead visit a node $(s, \nu X.\varphi)$ infinitely often instead. All plays which do not fulfill either of these conditions are (and should be) losing for the existential player, and consequently, they fail to show that the specification holds.

#### 3.3.3.1 Existential player has a winning strategy iff $\varphi$ is satisfied in $s$

Here we will argue in the first direction: assume that, for game node $(s, \varphi)$, the formula holds, and as such there is a winning strategy for the existential player from that node. We will then construct this strategy, $\sigma$, and argue for why all plays following it are won by the existential player.

The assumption that the formula holds is the basis for all below arguments, except for cases 1-3 where the formula is simply $\top$, $\bot$, or $P$.

Case 1: $(s, \top)$

In $(s, \top)$ we have $\mathcal{M}, s \models_{\mathcal{V}} \top$ and as such the strategy gives: $\sigma((s, \top)) = (s, \top)$

Case 2: $(s, \bot)$

Nodes $(s, \bot)$ are owned by the universal player and as such the strategy does not apply.

Case 3: $(s, P)$

For nodes $(s, P)$ we have $\mathcal{M}, s \models_{\mathcal{V}} P$ iff $s \in \mathcal{V}(P)$. If $s \in \mathcal{V}(P)$ then the node is owned by the existential player and as such the strategy gives: $\sigma((s, P)) = (s, P)$. Otherwise it is owned by the universal player and the strategy does not apply.

Case 4: $(s, \varphi_1 \wedge \varphi_2)$

Nodes $(s, \varphi_1 \wedge \varphi_2)$ are owned by the universal player and as such the strategy does not apply.

Case 5: $(s, \varphi_1 \vee \varphi_2)$

For nodes $(s, \varphi_1 \vee \varphi_2)$ we have $\mathcal{M}, s \models_{\mathcal{V}} \varphi_i$ for some $i \in \{1, 2\}$, therefore the strategy gives: $\sigma((s, \varphi_1 \vee \varphi_2)) = (s, \varphi_i)$.

Case 6: $(s, \Diamond\varphi)$

For nodes $(s, \Diamond\varphi)$ we have $\mathcal{M}, t \models_{\mathcal{V}} \varphi$ for some $t \in E_M(s)$, therefore the strategy gives: $\sigma((s, \Diamond\varphi)) = (t, \varphi)$.

Case 7: $(s, \Box\varphi)$

Nodes $(s, \Box\varphi)$ are owned by the universal player and as such the strategy does not apply.

Case 8: $(s, \nu X.\varphi)$

Nodes $(s, \nu X.\varphi)$ are owned by the universal player and as such the strategy does not apply.

Case 9: $(s, \mu X.\varphi)$

Nodes $(s, \mu X.\varphi)$ are owned by the universal player and as such the strategy does not apply.

Case 10: $(s, X)$

Nodes $(s, X)$ are owned by the universal player and as such the strategy does not apply.

Now that the strategy $\sigma$ has been defined we will argue for why all plays following $\sigma$ are winning: Let $\tau$ be an (infinite) play following $\sigma$. $\tau$ can follow only one of two patterns:

The first pattern entails $\tau$ eventually reaching a *leaf* node ($\{(s, \top), (s, \bot), (s, P)\}$) thereby infinitely visiting just that node. In this case, per the definition for the labeling function, the existential player wins iff $\mathcal{M}, s \models_{\mathcal{V}} \varphi$ (corresponding to either $(s, \top)$ or $(s, P)$ where $s \in \mathcal{V}(p)$). Therefore for this pattern $\mathcal{M}, s \models_{\mathcal{V}} \varphi$ implies that player $\exists$ wins the play $\tau$.

The second pattern entails $\tau$ infinitely avoiding such a node, consequently visiting a node containing a fixpoint infinitely often. In this case, again per the definition of the labeling function, the existential player wins iff the node visited contains a $\nu$-fixpoint (i.e. a node $(s, \nu X.\varphi)$). Only $\nu$-fixpoints allow for infinite unrolling, therefore for this pattern $\mathcal{M}, s \models_{\mathcal{V}} \varphi$ implies that the play $\tau$ is won by player $\exists$.

#### 3.3.3.2 If existential player wins $(s, \varphi)$ then the formula $\varphi$ is satisfied

Here we argue in the second direction: We assume that the existential player wins, from which we know that the existential player has a winning strategy, $\sigma$. Since we know that there is a winning strategy we know that it is possible to compute the certificate and from that certificate show that the formula is satisfied.

All below arguments depend on the assumption that the existential player wins the node, and as such a certificate, as defined in Section 3.3.2, can be extracted. More specifically, the argument presented is inductive and based on *walking* along a given certificate to extract the sub-formulae that hold. The exception to this are cases 1-3 where there is no assumption of the existential player winning because of these nodes being directly winning or losing.

Since the argument we present is based on induction, we will here define the base cases and the inductive hypothesis. The base cases consists of cases 1-3 and case 10. The induction hypothesis, then, is that from any given winning node $(s, \varphi)$ in the reduced game it is possible to show why $\mathcal{M}, s \models \varphi$ by *walking* along the certificate

starting in that node. We define *walking* along the certificate as following all paths in the certificate from $v$ until a base case is reached. When a base case is reached, it is possible to backtrack whilst recursively reducing each node to a node of the form $(s, \top)$. When the backtracking returns to the original node this *walk* along the certificate has then shown why $\varphi$, along with all the sub-formulae contained in the nodes in the certificate, hold.

Case 1: $(s, \top)$

This node is winning and the formula holds, therefore there is nothing to show.

Case 2: $(s, \bot)$

This node is, per the definition of the reduced game, losing. As it is losing it is not part of the certificate which suggests that the formula does not hold, which is correct.

Case 3: $(s, P)$

If $s \in \mathcal{V}(P)$ then this case is semantically identical to case 1, otherwise it is semantically identical to case 2.

Case 4: $(s, \varphi_1 \wedge \varphi_2)$

For the case where the formula is a conjunction one must show that $\mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \wedge \varphi_2$. As per the semantics, see Section 2.6.3, one must show that both $\mathcal{M}, s \models_{\mathcal{V}} \varphi_1$ and $\mathcal{M}, s \models \varphi_2$. Per definition of the certificate the successors of $(s, \varphi_1 \wedge \varphi_2)$ in the certificate are $(s, \varphi_1)$ and $(s, \varphi_2)$. By the inductive hypothesis, $\mathcal{M}, s \models_{\mathcal{V}} \varphi_i$ for all $i \in \{1, 2\}$ as required.

Case 5: $(s, \varphi_1 \vee \varphi_2)$

For the case where the formula is a disjunction one must show that $\mathcal{M}, s \models_{\mathcal{V}} \varphi_1 \vee \varphi_2$. As per the semantics, see Section 2.6.3, it suffices to show that $\mathcal{M}, s \models_{\mathcal{V}} \varphi_1$ or $\mathcal{M}, s \models \varphi_2$. Per definition of the certificate the successor of $(s, \varphi_1 \vee \varphi_2)$ in the certificate is either $(s, \varphi_1)$ or $(s, \varphi_2)$. By the inductive hypothesis, $\mathcal{M}, s \models_{\mathcal{V}} \varphi_i$ for some $i \in \{1, 2\}$ as required.

Case 6: $(s, \Diamond\varphi)$

For the case where the formula is of the form $\Diamond\varphi$ one must show that $\mathcal{M}, s \models_{\mathcal{V}} \Diamond\varphi$. As per the semantics, see Section 2.6.3, it suffices to show that $\mathcal{M}, t \models_{\mathcal{V}} \varphi$ for some $t \in E_M(s)$. Per definition of the certificate the successor of $(s, \Diamond\varphi)$ in the certificate is $(t, \varphi)$ for some $t \in E_M(s)$. By the inductive hypothesis, $\mathcal{M}, t \models_{\mathcal{V}} \varphi$ for some $t \in E_M(s)$ as required.

Case 7: $(s, \Box\varphi)$

For the case where the formula is of the form $\Box\varphi$ one must show that $\mathcal{M}, s \models_{\mathcal{V}} \Box\varphi$. As per the semantics, see Section 2.6.3, one must show that $\mathcal{M}, t \models_{\mathcal{V}} \varphi$ for all $t \in E_M(s)$. Per definition of the certificate the successors of $(s, \Diamond\varphi)$ in the certificate are $(t, \varphi)$ for all $t \in E_M(s)$. By the inductive hypothesis, $\mathcal{M}, t \models_{\mathcal{V}} \varphi$ for all $t \in E_M(s)$ as required.

Case 8: $(s, \nu X.\varphi)$

For the case where the formula is of the form $\nu X.\varphi$ one must show that $\mathcal{M}, s \models_{\mathcal{V}} \nu X.\varphi$. As per the semantics, see Section 2.6.3, one must show that $\exists U \subseteq 2^S, s \in U$ such that $\mathcal{M}, t \models_{\mathcal{V}[X/U]} \varphi$ for all $t \in U$. We define $U$ to be the set of all nodes $(t, \varphi)$ where $t \in S$ and $(t, \varphi)$ is contained in the certificate. Then, $s \in U$ since the successor of $(s, \nu X.\varphi)$ in the certificate is $(s, \varphi)$. For this definition of $U$ we must now show that $\mathcal{M}, t \models_{\mathcal{V}[X/U]} \varphi$ for all $t \in U$. This can be shown by a *walk* along the certificate where cases 1-7 hold per the inductive hypothesis, and for nodes encompassed by case 10, $(t', \varphi)$ we must show that $\mathcal{M}, t' \models_{\mathcal{V}[X/U]} X$, that is $t' \in U$. Since $(t', X)$ is contained in the certificate then its successor, $(t', \nu X.\varphi)$, is contained in the certificate and the successor of that node, $(t', \varphi)$, in turn is also in the certificate, as required. Therefore, in case 8, $\mathcal{M}, s \models_{\mathcal{V}} \nu X.\varphi$ as required.

Case 9: $(s, \mu X.\varphi)$

For the case where the formula is of the form $\mu X.\varphi$ one must show that $\mathcal{M}, s \models_{\mathcal{V}} \mu X.\varphi$. As per the semantics, see Section 2.6.3, one must show that $\forall U \subseteq 2^S$ if $s \notin U$ then $\exists t \in S : t \notin U$ and $\mathcal{M}, t \models_{\mathcal{V}[X/U]} \varphi$. Let $U$ be any such set, then $s \notin U$ and we must find some suitable $t$. Let $t = s$, we must now show that $\mathcal{M}, s \models \varphi$. This can be shown by a *walk* along the certificate starting in $(s, \varphi)$. Since the strategy, $\sigma$, used to define the certificate is a winning strategy this *walk* will encounter nodes encompassed by case 10 only finitely often. That is, at some point during the *walk* a part of the certificate will be reached where it is no longer possible to reach a node corresponding to case 10. Once such a part is reached then the *walk* is guaranteed to finish without seeing a node corresponding to case 10, that is to say, the induction will terminate in either a node corresponding to case 1 or case 3 (if case 3 is in the valuation). Thereby $\mathcal{M}, s \models_{\mathcal{V}} \varphi$ and consequently $\mathcal{M}, t \models_{\mathcal{V}} \mu X.\varphi$, as required.

Case 10: $(s, X)$

Case 10 only occurs in cases 8 and 9, where we already handle it. Therefore, there is nothing more to show here.

# 4

# Results

This chapter presents the results achieved during development. These results will include testing and runtime benchmarks of our game solver and Zielonka tree generation.

## 4.1 Zielonka trees

In this section we present the results from the implemented Zielonka tree structure and its generation.

### 4.1.1 Correctness

Correctness testing for the generation of Zielonka trees has been performed by comparison to known patterns and verification by hand. For example, the Zielonka trees for Parity, Büchi and generalized Büchi objectives branch in predictable ways, making them simple to verify by hand. Furthermore, Rabin and Street objectives have also been tested by hand and have given the correct results.

Since this thesis has a focus on Emerson-Lei games in particular, we also tested the general form of Emerson-Lei objectives to generate Zielonka trees. As the general form of Emerson-Lei objectives can be any kind of formula built from the allowed operations, we tested some arbitrary formulae for correctness. As an example, we tested generating Zielonka trees for the following formulae:

$$\varphi_1 = (\text{Inf } a \to \text{Inf } b) \wedge (\text{Inf } c \wedge \text{Fin } d) \wedge \text{Inf } e$$
$$\varphi_2 = (\text{Inf } a \to (\text{Inf } b \vee \text{Inf } c)) \wedge ((\text{Inf } d \vee \text{Inf } e) \to \text{Inf } a)$$

The generated Zielonka trees from the formulae can be seen in Figure 4.1, for $\varphi_1$, and Figure 4.2 for $\varphi_2$. These Zielonka trees generated by our implementation match the structures that we expected, indicating that they are correct.

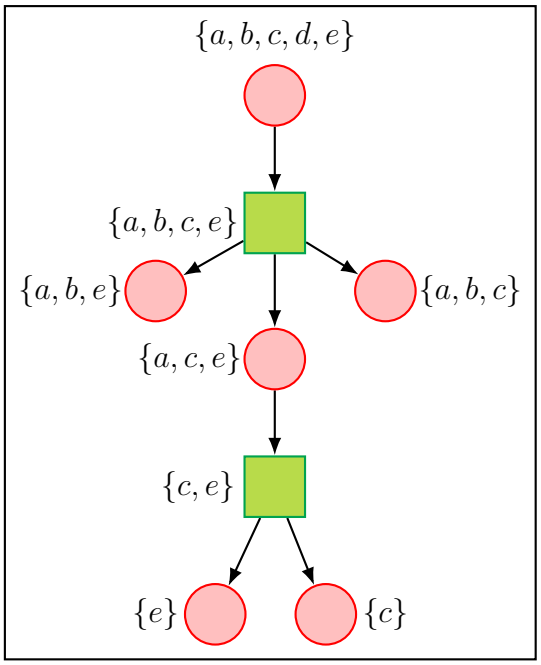


Figure 4.1: Zielonka tree for Emerson-Lei objective $\varphi_1$.

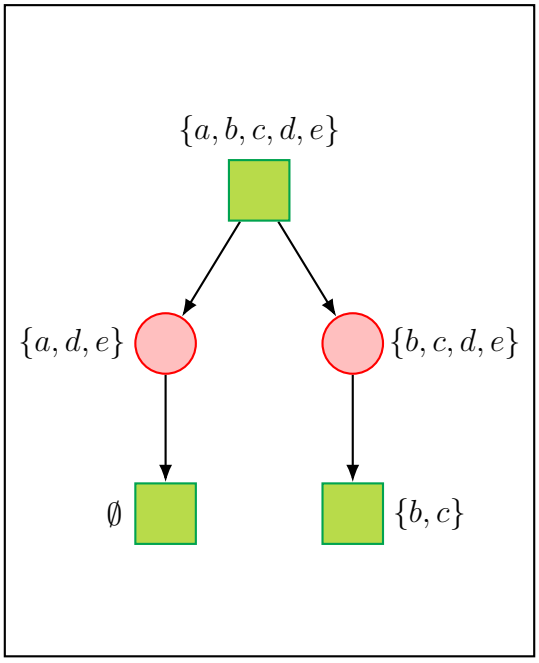


Figure 4.2: Zielonka tree for Emerson-Lei objective $\varphi_2$.

### 4.1.2 Generation performance

The performance of Zielonka tree generation can be benchmarked using various known types of conditions. We have chosen to benchmark with Rabin, Parity, Generalized Büchi and Streett objectives. These conditions can be scaled easily and give a rough estimate of the time it takes to generate Zielonka trees for different input sizes.

Figures 4.3 and 4.4 present the runtime performance of the Zielonka tree generation and the number of nodes that the Zielonka trees have for different types of conditions. These conditions are generated using a simple script which takes a number of colors or pairs $n$ as input and produces a formula corresponding to it.

## 4.2 Genie-EL

In Genie, we added the implementation for condition evaluation, the Zielonka and the Emerson-Lei game solving algorithm.

Some slight changes to the *CMakeLists.txt* file were also made to include our additions in the building process of the project.

## 4.3 FairSyn-EL

FairSyn-EL is the name of the fork of FairSyn that we have made additions to. Since most of our contributions were to Genie however, the only things we added

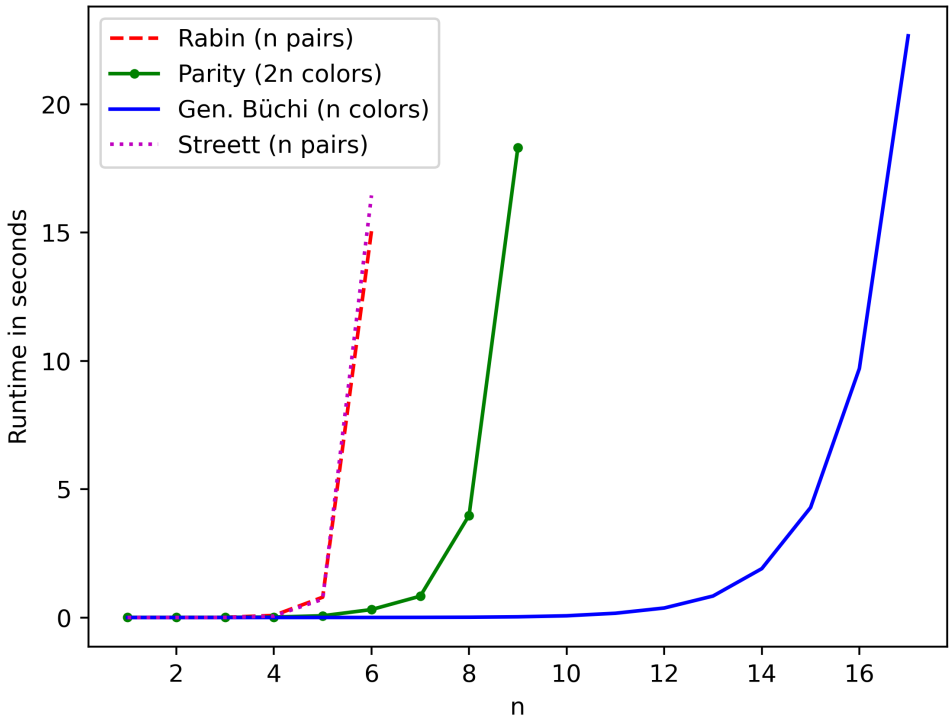


Figure 4.3: Graph showing the runtime of the Zielonka tree generation for Rabin, Parity, Generalized Büchi and Streett conditions.

to FairSyn-EL were our arena parsing functionality and a separate executable used to run games with Emerson-Lei objectives using our implemented game solving algorithm. The main entry point to the game solver is in FairSyn-EL, thus, this is the tool we have tested the most.

### 4.3.1 Testing

This section presents the general testing of features added to FairSyn-EL, such as solving correctness and runtime benchmarks.

#### 4.3.1.1 Correctness

To test that our implementation, presented in Listing 4, is correct, we have run the algorithm on various arena configurations. We have also compared the results from running the Rabin game solver, given in FairSyn, to our implementation. To ensure that not only Rabin games are being solved correctly, we have computed winning regions of games by hand and compared the results to those of our implementation. Some of the objectives we have tested include Büchi, generalized Büchi, Parity, Rabin and Streett, and all of them give correct results for the game arenas of limited sizes we have used.

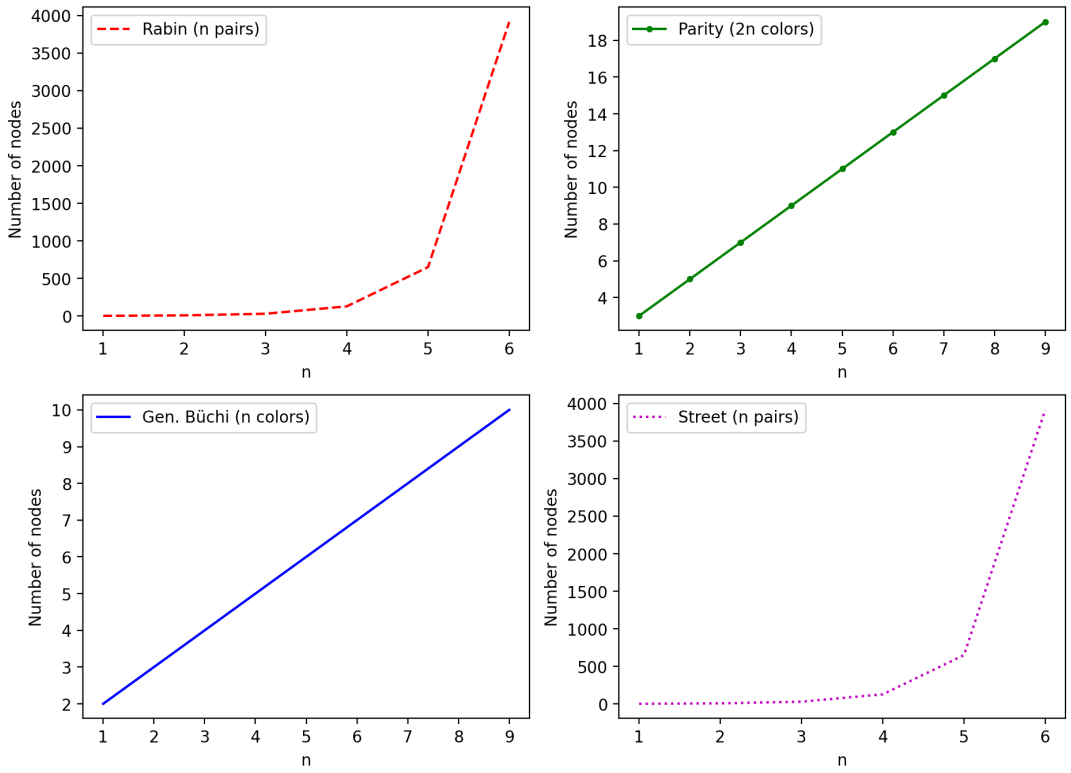


Figure 4.4: Number of nodes in the Zielonka tree with Rabin, Parity, Generalized Büchi, and Streett conditions.

#### 4.3.1.2 Hardware specifications

Benchmarking of the game solver was performed on a laptop with the hardware seen in Table 4.1. This hardware specification is not optimal for high performance, but since the benchmarks were done on the same machine the results are relative to each other.

| | |
|---|---|
| **Model** | HP ZBook Studio G4 |
| **Memory** | 16GiB SODIMM DDR4 Synchronous Unbuffered 2400 MHz (0.4ns) |
| **CPU** | Intel Xeon E3-1505M v6 (8) @ 4.000GHz |
| **GPU** | NVIDIA Quadro M1200 Mobile |
| **OS** | Manjaro Linux x86_64 |
| **Kernel** | 6.6.26-1-MANJARO |

Table 4.1: Benchmark hardware specifications

#### 4.3.1.3 Benchmarks

This section presents the benchmarks we have run to test the runtime performance of the solver. The differences in performance compared to FairSyn with Genie are also given. In the end we present the runtime performance for some general Emerson-Lei objectives. All of the arenas we chose to benchmark our solver on were created by hand, except for the arena in Figure 4.5, which was taken from the Wikipedia page for Parity games [18]. The arenas we created test the runtime of the game solver for both increasing amount of colors and game nodes.

For the Parity game with arena seen in Figure 4.5, taken from [18], it takes the solver approximately 32 seconds to solve the game.

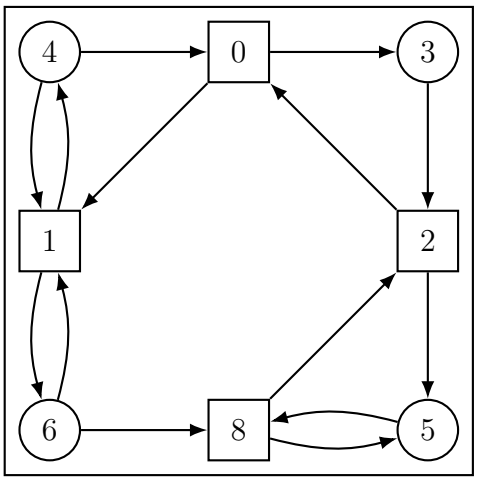


Figure 4.5: Example arena for benchmarking with Parity condition [18].

In Figure 3.2 we can add two more colors and get the arena in Figure 4.6. We tested this arena with Rabin, Streett and Generalized Büchi conditions and the results can be seen in Table 4.2. Tests on other arenas were also done, see Figures 4.7, 4.8, 4.9 and 4.10. These tests used the same objectives and the results can be seen in Tables 4.3, 4.4, 4.5 and 4.6.

The performance difference between our implementation and FairSyn throughout the different tests can be seen in Table 4.7. Benchmarks of general Emerson-Lei conditions can be seen in Tables 4.8 and 4.9, for arenas in Figures 4.11 and 4.12 respectively. The objectives tested in those tables have their Zielonka trees depicted in Figures 4.1 and 4.2 respectively.

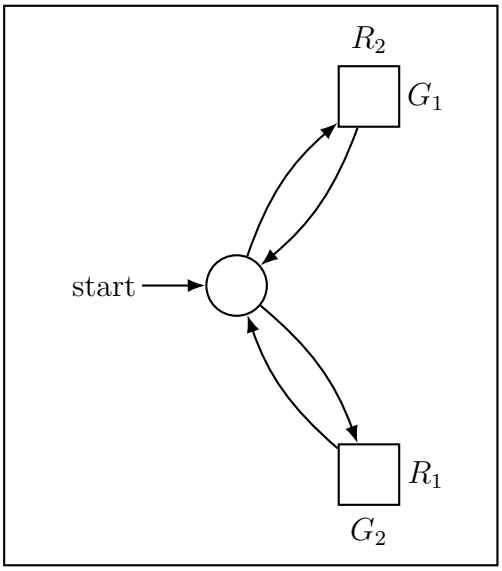


Figure 4.6: Benchmarking arena $A_1$.

| **Objective** | **Formula** | **Runtime** |
|---|---|---|
| Rabin | $\bigvee_{i\in\{1,2\}}(\text{Inf } R_i \wedge \text{Fin } G_i)$ | 0.31s |
| FairSyn (CUDD) | -\|\|- | 0.10s |
| FairSyn (Sylvan) | -\|\|- | 0.09s |
| Streett | $\bigwedge_{i\in\{1,2\}}(\text{Inf } R_i \rightarrow \text{Inf } G_i)$ | 0.42s |
| Generalized Büchi | $\bigwedge_{i\in\{1,2\}}(\text{Inf } R_i \wedge \text{Inf } G_i)$ | 0.04s |

Table 4.2: Benchmarking for arena $A_1$ in Figure 4.6.

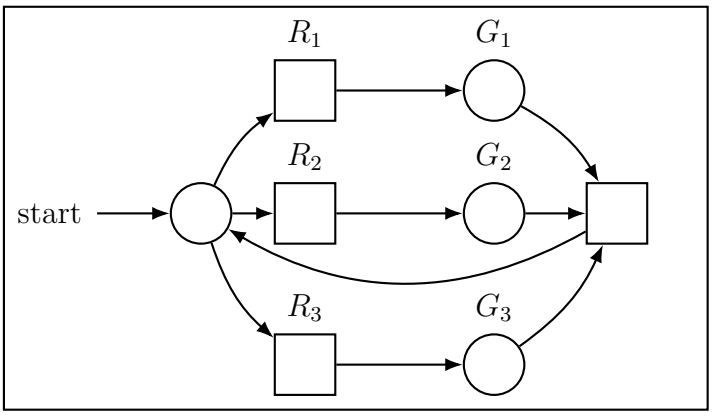


Figure 4.7: Benchmarking arena $A_2$.

| **Objective** | **Formula** | **Runtime** |
|---|---|---|
| Rabin | $\bigvee_{i\in\{1,2,3\}}(\text{Inf } R_i \wedge \text{Fin } G_i)$ | 3.39s |
| FairSyn (CUDD) | -\|\|- | 1.75s |
| FairSyn (Sylvan) | -\|\|- | 2.46s |
| Streett | $\bigwedge_{i\in\{1,2,3\}}(\text{Inf } R_i \rightarrow \text{Inf } G_i)$ | 7.21s |
| Generalized Büchi | $\bigwedge_{i\in\{1,2,3\}}(\text{Inf } R_i \wedge \text{Inf } G_i)$ | 0.06s |

Table 4.3: Benchmarking for arena $A_2$ in Figure 4.7.

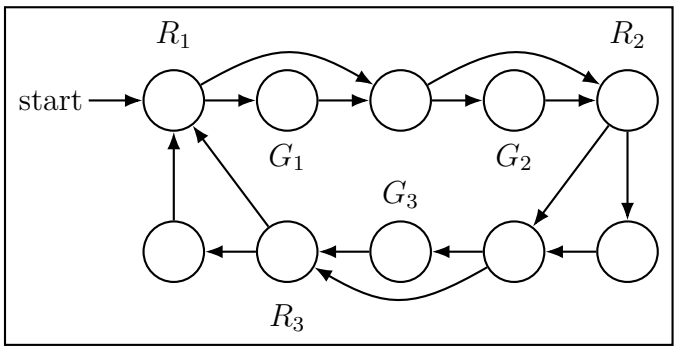


Figure 4.8: Benchmarking arena $A_3$.

| **Objective** | **Formula** | **Runtime** |
|---|---|---|
| Rabin | $\bigvee_{i\in\{1,2,3\}}(\text{Inf } R_i \wedge \text{Fin } G_i)$ | 7.65s |
| FairSyn (CUDD) | -\|\|- | 1.55s |
| FairSyn (Sylvan) | -\|\|- | 2.38s |
| Streett | $\bigwedge_{i\in\{1,2,3\}}(\text{Inf } R_i \rightarrow \text{Inf } G_i)$ | 14.8s |
| Generalized Büchi | $\bigwedge_{i\in\{1,2,3\}}(\text{Inf } R_i \wedge \text{Inf } G_i)$ | 0.06s |

Table 4.4: Benchmarking for arena $A_3$ in Figure 4.8.

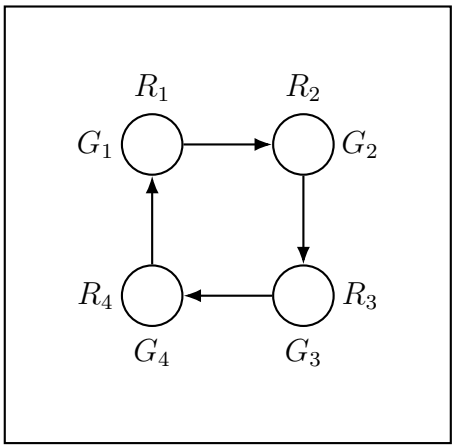


Figure 4.9: Benchmarking arena $A_4$.

| **Objective** | **Formula** | **Runtime** |
|---|---|---|
| Rabin | $\bigvee_{i\in\{1,2,3,4\}}(\text{Inf } R_i \wedge \text{Fin } G_i)$ | 6.75s |
| FairSyn (CUDD) | -\|\|- | 1.35s |
| FairSyn (Sylvan) | -\|\|- | 0.77s |
| Streett | $\bigwedge_{i\in\{1,2,3,4\}}(\text{Inf } R_i \rightarrow \text{Inf } G_i)$ | 6.70s |
| Generalized Büchi | $\bigwedge_{i\in\{1,2,3,4\}}(\text{Inf } R_i \wedge \text{Inf } G_i)$ | 0.06s |

Table 4.5: Benchmarking for arena $A_4$ in Figure 4.9.

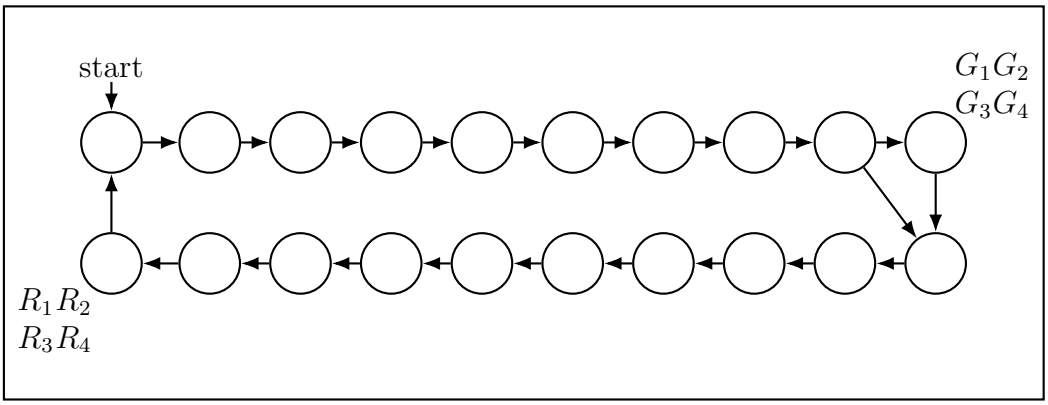


Figure 4.10: Benchmarking arena $A_5$.

| **Objective** | **Formula** | **Runtime** |
|---|---|---|
| Rabin | $\bigvee_{i\in\{1,2,3,4\}}(\text{Inf } R_i \wedge \text{Fin } G_i)$ | 80.43s |
| FairSyn (CUDD) | -\|\|- | 18.49s |
| FairSyn (Sylvan) | -\|\|- | 3.63s |
| Streett | $\bigwedge_{i\in\{1,2,3,4\}}(\text{Inf } R_i \rightarrow \text{Inf } G_i)$ | 79.73s |
| Generalized Büchi | $\bigwedge_{i\in\{1,2,3,4\}}(\text{Inf } R_i \wedge \text{Inf } G_i)$ | 0.15s |

Table 4.6: Benchmarking for arena $A_5$ in Figure 4.10.

| **Arena** | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
|---|---|---|---|---|---|
| **Implementation runtime** | 0.31 s | 3.39 s | 7.65 s | 6.75 s | 80.43 s |
| **FairSyn (CUDD) runtime** | 0.10 s | 1.75 s | 1.55 s | 1.35 s | 18.49 s |
| **FairSyn (Sylvan) runtime** | 0.09 s | 2.46 s | 2.38 s | 0.77 s | 3.63 s |
| **FairSyn (CUDD) speedup** | 309% | 194% | 494% | 500% | 435% |
| **FairSyn (Sylvan) speedup** | 344% | 138% | 321% | 876% | 2215% |
| **FairSyn (CUDD) mean speedup** | 386% | | | | |
| **FairSyn (Sylvan) mean speedup** | 779% | | | | |

Table 4.7: Performance difference for Rabin objectives between our implementation and FairSyn for both CUDD and Sylvan.

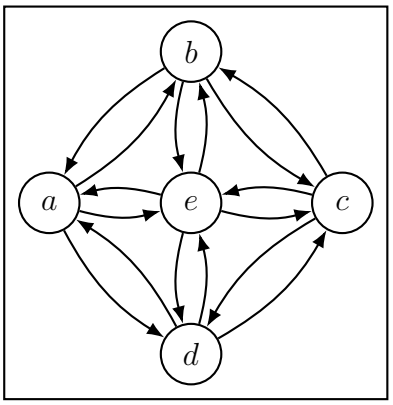


Figure 4.11: Emerson-Lei benchmarking arena $A_6$.

| **Formula** | **Runtime** |
|---|---|
| $(\text{Inf } a \rightarrow \text{Inf } b) \wedge (\text{Inf } c \wedge \text{Fin } d) \wedge \text{Inf } e$ | 0.61 s |
| $(\text{Inf } a \rightarrow (\text{Inf } b \vee \text{Inf } c)) \wedge ((\text{Inf } d \vee \text{Inf } e) \rightarrow \text{Inf } a)$ | 0.11 s |

Table 4.8: Benchmarking for arena $A_6$ in Figure 4.11.

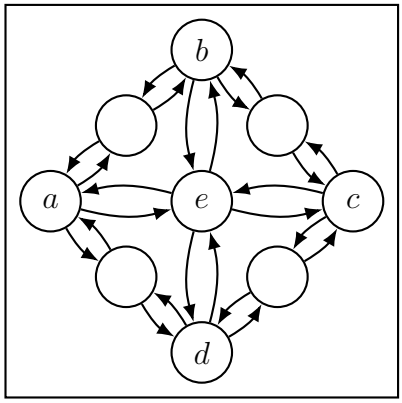


Figure 4.12: Emerson-Lei benchmarking arena $A_7$.

| **Formula** | **Runtime** |
|---|---|
| $(\text{Inf } a \to \text{Inf } b) \wedge (\text{Inf } c \wedge \text{Fin } d) \wedge \text{Inf } e$ | 1.19 s |
| $(\text{Inf } a \to (\text{Inf } b \vee \text{Inf } c)) \wedge ((\text{Inf } d \vee \text{Inf } e) \to \text{Inf } a)$ | 0.13 s |

Table 4.9: Benchmarking for arena $A_7$ in Figure 4.12.

## 4.4 Reduction

In Section 3.3 we presented a reduction from the alternation-free fragment of $\mu$-calculus to Büchi games. We then defined certificates in Section 3.3.2. Using this definition we then presented an argument for the correctness of our reduction in Section 3.3.3.

# 5

# Discussion and Conclusion

In this chapter we discuss the results of this project, the improvements to be made, limitations that we have encountered, and lastly we present our conclusions.

## 5.1 Discussion

From the testing and benchmarking presented in Chapter 4 we can conclude that the game solving functionality works but with suboptimal efficiency when it comes to runtime performance. Solving Emerson-Lei games in general is of exponential time complexity with respect to $|C|$ and without any memoization it is a reasonable result. If some kind of memoization were to be used, it would reduce the number of duplicate computations, and the runtime would subsequently decrease. Genie has a form of this memoization, it keeps an array of a specified number of previous computations. If a problem is precomputed, Genie will use it straight away and otherwise it will be computed and put in the array.

The generation of Zielonka trees is of exponential time complexity with respect to $|C|$ and is therefore slow for color sets of large caridinality. Zielonka trees for known types of objectives, such as generalized Büchi or Parity, can be generated from a template, which reduces the time complexity. For the general case of Emerson-Lei objectives however, there are no fixed templates to use and they will therefore always be relatively slow to generate.

The benchmarks which can be seen in Section 4.3.1.3 give a good understanding of how our implementation compares to FairSyn. With the same configuration performing the same task, FairSyn is on average almost 4 times as fast as our implementation. Comparing the runtime to the parallel behaviour of FairSyn with Sylvan, we see that the difference is even larger; almost 8 times faster on average. Although our implementation is not as fast as FairSyn and Genie, these results are still reasonable. Without optimization and parallelism, we could not expect to have as good performance as that of an established tool. Genie and FairSyn are also created for the sole purpose of solving Rabin games, and because of that they can have a hard-coded behaviour in their fixpoint algorithm. For our implementation, the fixpoint ordering has to be decided at runtime depending on the nodes in a Zielonka tree. This makes our algorithm less efficient and that is also possibly a reason why our implementation is slower.

One thing that was interesting was that, for some arenas, our implementation outperformed FairSyn when the `M` parameter, which specifies the maximal amount of memory to use, was increased. Our assumption is that when the parameter is increased, the algorithm uses an unnecessary amount of memory to solve the game, thereby increasing the runtime. Setting `M` to 12 actually crashed the benchmarking machine for one arena. This is most likely due to it attempting to allocate too much memory.

We created the benchmarking arenas, see Section 4.3.1.3, to test out how many colors our implementation would be able to handle in a reasonable amount of time. The increased amount of nodes and colors are also interesting to analyze. Looking at how our implementation performs for an increased amount of nodes we can see in Tables 4.5 and 4.6 that the runtime for the Rabin objectives are very different. This result suggests that the amount of game nodes has a quite large impact. Increasing the amount of colors also increases the runtime of course, this can be seen in the benchmark results.

Another thing which is interesting regarding the runtime of different objectives is that Rabin objectives seem to be somewhat faster than Street objectives in some cases. This might be due to the Rabin objectives not having solutions for that particular arena.

When it comes to benchmarking the runtime of game solution for general Emerson-Lei objectives, we did not have any solvers to compare against. This fact made it difficult to draw conclusions regarding the performance. We did however include benchmarks for two arenas and two objectives in Tables 4.8 and 4.9. Since Emerson-Lei objectives are a superset of the other objectives we tested, their benchmark are as relevant as for the general Emerson-Lei objectives.

The reduction that we give in Section 3.3.1 and argue for in Section 3.3.3 was brought forth independently but has a similar formulation to the reduction given by Martin Leucker [1]. While we knew that this reduction had already been given, we wanted to see what differences would arise in comparison to the original reduction. Therefore, we defined the reduction independently from Leucker's work and argued, also independently, for its correctness.

One notable difference between our reduction and the one Leucker gives is that our reduction was developed with Emerson-Lei games in mind, and as such the resulting game is directly given in the notation we defined for Emerson-Lei games. One benefit of this is that manually entering these games as input to our game solver would be quite easy. It would also be relatively easy to implement a tool for automatically performing this reduction.

Another difference is that Leucker's reduction applies to a more general form of $\mu$-calculus than ours. In his reduction he included actions $a$ [7] which in turn led to the use of additional terms, such as $\langle a \rangle \varphi$ and $[a]\varphi$. For our purpose it was good enough to leave out the actions from the notation. Semantically, this would mean that all edges from a certain node in the model encode the same action. In terms of expressiveness this is slightly restrictive since it excludes the possibility of edges

encoding a different action compared to another edge from the same node.

## 5.2 Improvements and future work

As can be seen in Figures 4.3 and 4.4, the runtime of the generation of a Zielonka tree increases drastically with an increase in the number of colors, but the number of nodes does not necessarily follow this same pattern. The number of nodes for the Parity and Generalized Büchi objectives increases linearly in relation to the number of colors, while the Rabin and Street conditions have a more drastic growth. Since both parity and Rabin games have known Zielonka tree structures, we want the `generate` method to have an additional parameter `mode` which specifies a known game condition and generates the tree from such a template. For the Parity objectives, we already have a prototype, which after a small amount of testing looks very promising in regards to decreasing the runtime of generation.

This change will greatly increase the efficiency of the generation because there will not be any need to repeatedly look through the powerset of colors, which is the part of the generation which most likely contributes the most to the slow down.

Another change that could be made is to reduce the number of iterations over the powerset of colors for each node. At this moment, the whole powerset is iterated over for each node to check for a valid child node from the parent node. One way we might improve this is to only loop over the set difference $C \setminus C_r$ where $C$ is the set of all colors and $C_r$ is the set of the colors which have already been removed. We could change the `ZielonkaNode` to not only include which colors were removed from the parent node but also from the root node, which includes all colors. After that, we could add another loop in the generation function to skip all the subsets which contain already removed colors.

The way we store the colors in each node in the Zielonka tree is probably also suboptimal. As colors are represented by vectors of Boolean values, the cardinality of these vectors is always fixed to $|C|$, where $C$ is the set of all colors. This leads to increased iteration runtime and memory use. Changing the format of the colors to include only the boolean values representing colors which satisfy Inf $c$, for $c \in C$, would significantly reduce this runtime. This would however demand that we change the structure of the evaluation functionality.

Something which would likely have a major impact on performance, would be implementing parallelism using Sylvan. As we can see in Section 4.3.1.3, Sylvan gives a drastic speedup compared to CUDD, and would therefore, most likely, produce similar results for our implementation. For even further speedup, memoization could be used to reduce the amount of duplicate computations. This could be implemented in a similar way to how Genie already does it.

We implemented Algorithm 2 in a way which was not based on any correctness proof. To completely verify that the behavior in [3] is achieved, a correctness proof of our implementation would be needed. We deem the implementation to be good enough from the testing described in Section 4.3.1.1, but as a future improvement,

a correctness proof should be provided.

One useful future endeavor, once our implementation is proven to be correct and optimized, is for the functionality we have developed to be integrated into Genie. This would make our implementation as easily accessible for game solving as Genie is at the moment.

An improvement that we would like to see in the future is the implementation of an automaton to create the games, instead of using the arena input format described in Section 3.2. Genie and FairSyn already have a way to do this, which makes it possible to read game inputs from the HOA [19] (Hanoi Omega-Automaton) format. Since there are a lot of existing game inputs in this format already, it would make thorough benchmarking more easily accessible.

## 5.3 Limitations

Something we should have noticed early on in the project was that MascotSDS was not the most suitable program to use for our purpose. It solves Rabin games but in a way which does not help us much. After consultation with the contributors of Genie, FairSyn and MascotSDS, we realized that Genie and FairSyn were more suitable for us to use. Due to this mistake, we lost some time researching MascotSDS in the beginning of the project which could instead have been used to understand the mechanisms of Genie for example. However, when we were informed of our mistake, the pace picked up as we were able to make better use of our time.

In the beginning of this thesis project, we predicted that the game solver would take less time to implement and that it would leave more time to investigate strategy extraction. This prediction was however too ambitious and we realized that strategy extraction would be too complicated to dive further into.

Parallel solution was also something that we wanted to do from the start, but as mentioned, trying to understand the regular behaviour of Genie and FairSyn required more effort than predicted. Parallelism became a lower priority for that reason and is now an idea for future work as an optimization.

When developing the generation for the Zielonka trees, we aimed for an implementation which worked in the cases which we were to use. This meant that we did not prioritize any formal correctness proofs for the algorithm. We therefore mark this as an improvement for the future to give more confidence in the implementation itself.

## 5.4 Conclusion

As a conclusion of this project we can say that we succeeded in developing the first symbolic game solver for Emerson-Lei games. Through testing we managed to verify that the solver produces correct results for various game configurations. The runtime performance, however, is suboptimal but can be improved with a few optimizations.

We revisited reductions from model checking of the alternation-free fragment of $\mu$-calculus to Büchi games, using the notation we gave for Emerson-Lei games. With this we showed how one could solve these model checking problems using our solver.

From these conclusions, we have successfully answered the research questions we stated in Section 1.6.

# Bibliography


[1] M. Leucker, "Model checking games for the alternation-free $\mu$-calculus and alternating automata," in *Logic for Programming and Automated Reasoning*, H. Ganzinger, D. McAllester, and A. Voronkov, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 77–91, ISBN: 978-3-540-48242-0.

[2] *The Reactive Synthesis Competition | www.syntcomp.org — syntcomp.org*, `http://www.syntcomp.org/`, [Accessed 02-06-2024].

[3] D. Hausmann, M. Lehaut, and N. Piterman, "Symbolic solution of emerson-lei games for reactive synthesis," in *Foundations of Software Science and Computation Structures - 27th International Conference, FoSSaCS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, N. Kobayashi and J. Worrell, Eds., ser. Lecture Notes in Computer Science, vol. 14574, Springer, 2024, pp. 55–78. DOI: `10.1007/978-3-031-57228-9\_4`. [Online]. Available: `https://doi.org/10.1007/978-3-031-57228-9%5C_4`.

[4] E. Allen Emerson and C.-L. Lei, "Modalities for model checking: Branching time logic strikes back," *Science of Computer Programming*, vol. 8, no. 3, pp. 275–306, 1987, ISSN: 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(87)90036-0`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0167642387900360`.

[5] S. Safra and M. Y. Vardi, "On -automata and temporal logic," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '89, Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 127–137, ISBN: 0897913078. DOI: `10.1145/73007.73019`. [Online]. Available: `https://doi.org/10.1145/73007.73019`.

[6] E. Grädel, W. Thomas, and T. Wilke, Eds., *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, vol. 2500, Lecture Notes in Computer Science, Springer, 2002, ISBN: 3-540-00388-6. DOI: `10.1007/3-540-36387-4`. [Online]. Available: `https://doi.org/10.1007/3-540-36387-4`.

[7] D. Kozen, "Results on the propositional µ-calculus," *Theoretical Computer Science*, vol. 27, no. 3, pp. 333–354, 1983, Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982, ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(82)90125-6`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0304397582901256`.

[8] P. Hunter and A. Dawar, "Complexity bounds for regular games," in *Mathematical Foundations of Computer Science 2005*, J. Jdrzejowicz and A. Szepietowski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 495–506, ISBN: 978-3-540-31867-5.

[9] *Genie*. [Online]. Available: `https://gitlab.mpi-sws.org/mrychlicki/genie`.

[10] *Fairsyn*. [Online]. Available: `https://gitlab.mpi-sws.org/kmallik/fairsyn`.

[11] *Mascotsds*. [Online]. Available: `https://gitlab.mpi-sws.org/kmallik/mascotsds`.

[12] R. Majumdar, K. Mallik, M. Rychlicki, A.-K. Schmuck, and S. Soudjani, "A flexible toolchain forăsymbolic rabin games under fair andăstochastic uncertainties," in *Computer Aided Verification*, C. Enea and A. Lal, Eds., Cham: Springer Nature Switzerland, 2023, pp. 3–15, ISBN: 978-3-031-37709-9.

[13] *Cudd*. [Online]. Available: `https://github.com/ivmai/cudd`.

[14] T. van Dijk and J. van de Pol, "Sylvan: Multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 675–696, Nov. 2017, ISSN: 1433-2787. DOI: `10.1007/s10009-016-0433-2`. [Online]. Available: `https://doi.org/10.1007/s10009-016-0433-2`.

[15] W. Zielonka, "Infinite games on finitely coloured graphs with applications to automata on infinite trees," *Theoretical Computer Science*, vol. 200, no. 1, pp. 135–183, 1998, ISSN: 0304-3975. DOI: `https://doi.org/10.1016/S0304-3975(98)00009-7`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0304397598000097`.

[16] *Genie-el*. [Online]. Available: `https://github.com/chrilars/genie-EL`.

[17] *Fairsyn-el*. [Online]. Available: `https://github.com/chrilars/fairsyn-EL`.

[18] Wikipedia, *Parity game — Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/w/index.php?title=Parity%20game&oldid=1188919474`, [Online; accessed 27-May-2024], 2024.

[19] T. Babiak, F. Blahoudek, A. Duret-Lutz, *et al.*, "The hanoi omega-automata format," Jul. 2015, pp. 479–486, ISBN: 978-3-319-21689-8. DOI: `10.1007/978-3-319-21690-4_31`.

# A

# Appendix 1

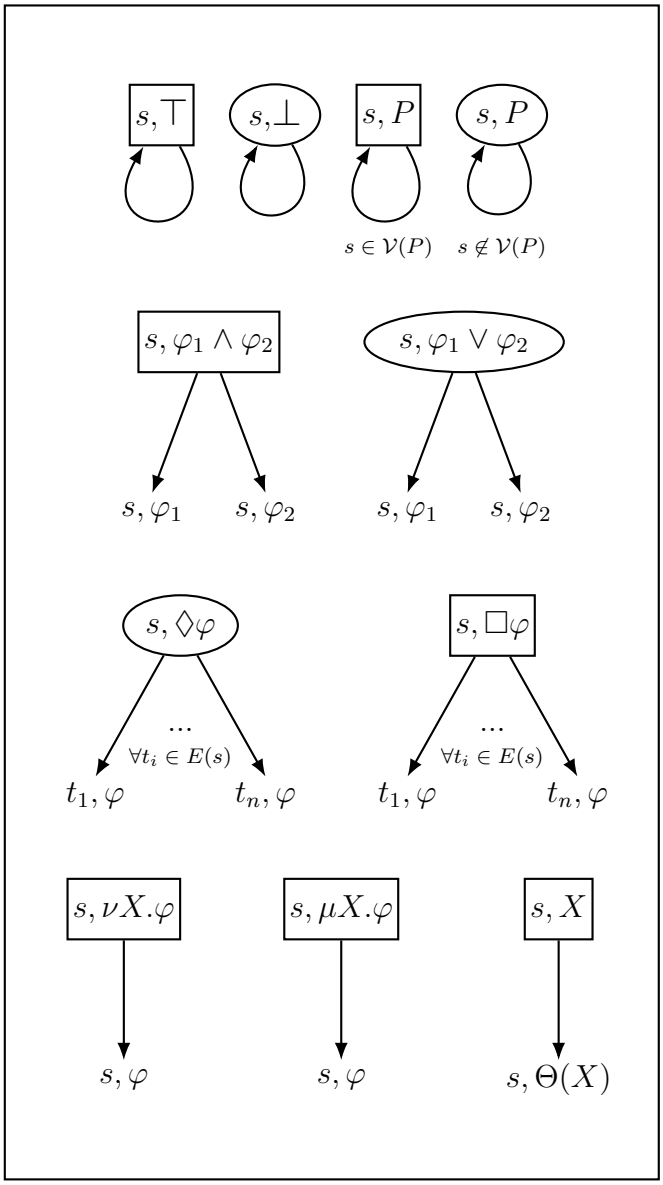


Figure A.1: Visual representation of the construction of game nodes for the $\mu$-calculus game reduction.

```
void ZielonkaTree::generate() {
    std::queue<ZielonkaNode*> q;
    q.push(root);
    std::vector<std::vector<bool>> ps = ELHelpers::powerset(root->label.size());
    std::sort(ps.begin(), ps.end(), cmp_descending_count_true);
    size_t order = root->order + 1;
    std::vector<std::vector<bool>> seen_from_parent{};
    while (!q.empty()) {
        seen_from_parent.clear();
        ZielonkaNode* current = q.front();
        q.pop();
        total_nodes++;
        for (size_t i = 0; i < ps.size(); ++i) {
            std::vector<bool> color_set = ps[i];
            if (!ELHelpers::proper_subset(color_set, current->label))
                continue;
            bool seen = false;
            for (const auto& s : seen_from_parent) {
                if (ELHelpers::proper_subset(color_set, s)) {
                    seen = true;
                    break;
                }
            }
            if (seen) continue;
            if (evaluate_phi(color_set) != current->winning) {
                ZielonkaNode *child_zn = new ZielonkaNode {
                    .children = {},
                    .child_differences = {},
                    .parent = current,
                    .parent_order = current->order,
                    .label = color_set,
                    .level = current->level + 1,
                    .order = order++,
                    .winning = !(current->winning)
                };
                current->child_differences.push_back(
                    ELHelpers::label_difference(current->label, color_set)
                );
                seen_from_parent.push_back(color_set);
                current->children.push_back(child_zn);
                q.push(child_zn);
            }
        }
        if (current->children.empty()) leaves++;
    }
}
```

Listing 6: The generate method of the ZielonkaTree class.

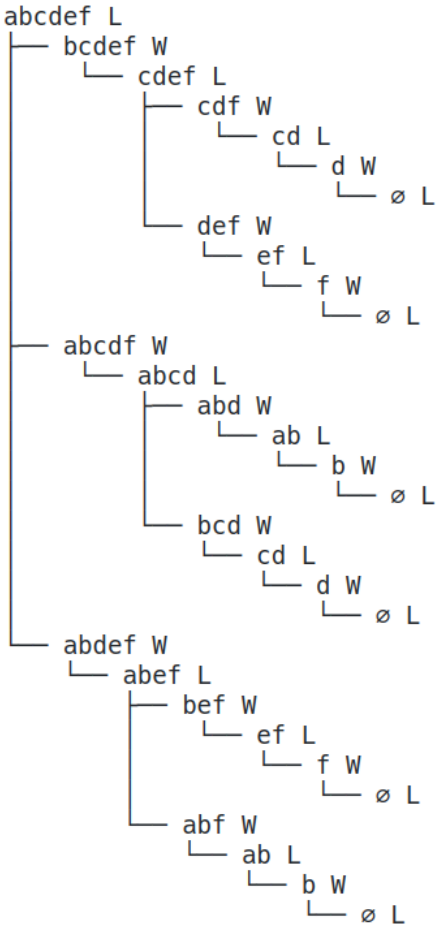


Figure A.2: ASCII representation of the Zielonka tree generated from the Rabin objective with 3 Rabin pairs.

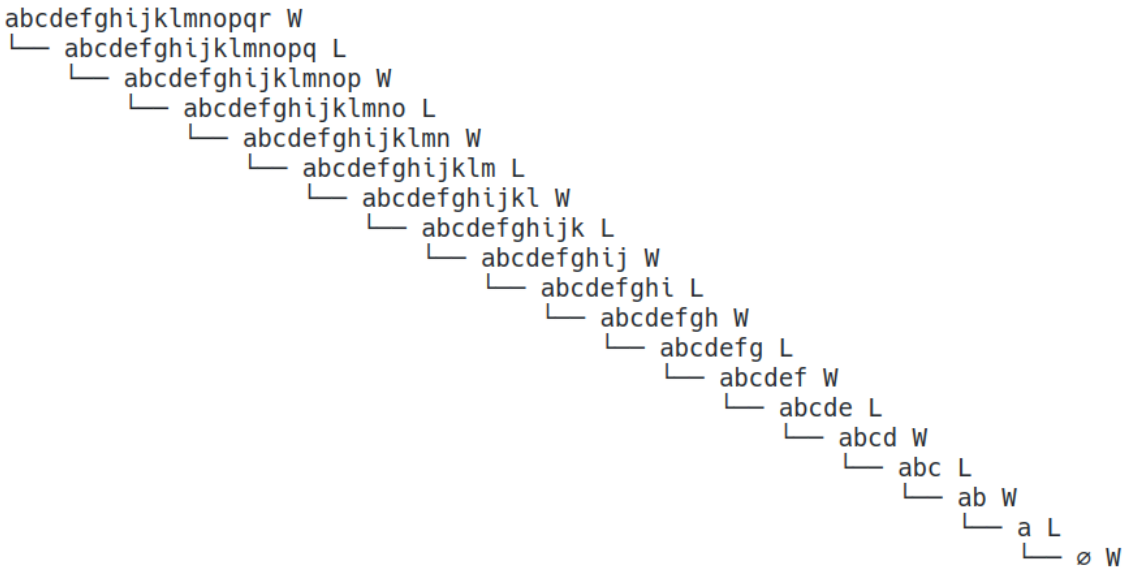


Figure A.3: ASCII representation of the Zielonka tree generated from the Parity conditions with 18 colors.