University of Padua

Department of Mathematics, Tullio Levi-Civita

Master Thesis in Computer Science

A Local Algorithm for Systems of Fixpoint Equations: Study and Implementation

Supervisor
Prof. Paolo Baldan
University of Padua

Co-supervisor
Prof. Tommaso Padoan
University of Trieste

Master Candidate
Alessandro Flori

December 2023

Acknowledgements

Ci tengo innanzitutto a ringraziare il Professor Baldan e il Professor Tommaso Padoan, per avermi seguito e spronato durante questi ultimi frenetici mesi. Ringrazio la mia famiglia e parenti italiani e non, che mi hanno sempre supportato. Ringrazio tutti gli amici che mi hanno accompagnato lungo questo percorso, in particolare i compagni di corso, senza alcuni dei quali non sarei qua. Ringrazio Ilaria per la sua incredibile pazienza. Ringrazio nonno Michael, che mi ha aiutato per la stesura di questa tesi. Un ringraziamento va anche a nonna Liliana, che ha sicuramente aiutato Michael ad aiutare me. Infine, dedico questa tesi a nonna Tonina, che si è sempre mostrata entusiasta almeno quanto me per i traguardi da me raggiunti.

Alessandro Flori

Abstract

Many verification tasks, such as model checking of various kinds of specification logics, can be expressed via a system of (mixed) fixpoint equations over complete lattices. Recent contributions showed that the solution of systems of fixpoint equations have a game-theoretical characterisation in terms of a parity game. We call this type of parity game, built on top of a system of fixpoint equations, the powerset game.

This game representation allows us to develop a local algorithm: we are particularly interested in finding the winner of the game from a specific position of the game graph (for example, for model checking, whether a state satisfies a formula). A naive implementation of the local algorithm could lead to the exploration of a uselessly large part of the game graph. We can reduce the number of moves, consequently shortening game play, thanks to the notion of selection, which intuitively selects a subset of best moves amongst all the possible moves.

The main contribution of this thesis is an algorithmic overview of the powerset game, and a tool written in the language Rust. This tool is able to check whether a player has a winning strategy in the powerset game from a specific position, which formally corresponds to checking whether a lattice element is included in the solution of the system of fixpoint equations. It is our concern to ensure that the tool interoperates with the like of mCRL2 [12] and Oink [6]: this allows us to access a wide pool of examples as well as to compare our tool with the state of the art.

Contents

List of Figures

1 Introduction

Systems of fixpoint equations over complete lattices with mixed least and greatest fixpoints are ubiquitous in formal analysis and verification. Notable examples come from the area of model checking, where we can express liveness/reachability properties as least fixpoints and safety/invariant properties as greatest fixpoints. A type of logic for model checking that we will use in this thesis is the μ-calculus [14]. Behavioural equivalence is another field where fixpoint equations appear. For example, bisimilarity can be characterised as a greatest fixpoint of a suitable operator over the lattice of binary relations on the set of states. Other prominent examples come from the area of static analysis, and more specifically from program analysis [19] and abstract interpretation [4]. In these instances, the control flow graph of a program is used to generate a set of constraints specifying how the information of interest, at the different program points, is interrelated. The constraints can then be viewed as a system of fixpoint equations, with least or greatest fixpoints. In almost all of these applications, the functions involved are monotone, and the domain of interest is a complete lattice. In this setting the existence of least and greatest fixpoints is guaranteed by the Knaster-Tarski's fixpoint theorem [21]. A theoretical game is devised in [3] in order to characterise solutions of systems of fixpoint equations over continuous lattices, in terms of a parity game, between the existential player and the universal player. This type of game is called the fixpoint game. This game characterisation opens the way to the development of both global and local algorithms for solving systems of fixpoint equations.

The global algorithm described in [3], uses the notion of progress measures, first introduced in [13], adapted to the setting of fixpoint games. Progress measures are a tool originally devised for solving parity games and, intuitively, they are witnesses of the existence of a winning strategy for the existential player of a parity game. More formally, in the setting of a fixpoint game played on a system of fixpoint equations E, over a continuous lattice L, the existence of a progress measure from a position of the existential player intuitively means that the lattice element represented by such position is below E's solution. Moreover, [3] introduces the notion of *selection*, a way to restrict the existential player moves. These selections can then be conveniently represented by a symbolic formula, exploiting a characteristic of the set of moves of the existential player, in the line of [5]. Mazzocchin, in [16], developed a tool, called PMModelchecker [15], that computes global solutions of systems of fixpoint equations, using the theory mentioned above.

We can further generalize the definition of fixpoint game, in order to be able to characterise solutions of system of fixpoint equations over complete lattices. This type of game is shown in [2], and it is called a powerset game. Moreover, a local algorithm for a powerset game is shown in [2], that is, we check whether the existential player wins from a specific position of the game graph. For instance, in the case of the μ-calculus, rather than computing the set of states satisfying some formula φ, one could be interested in checking whether a specific state satisfies or not φ. Similarly, in the case of behavioural equivalences, rather than computing the full behavioural relation, one could be interested in determining whether two specific states are equivalent.

Our main contribution is a refinement of the local algorithm in [17], which in turn improves the proposal by [2]. This algorithm, given a system of fixpoint equations E over a complete lattice and the corresponding set of symbolic $\exists$-moves, is able to verify whether a lattice element is below the solution of E, with respect to some ordering. The local algorithm is based on an algorithm for model checking the μ-calculus, by [20]. Moreover, we no longer need the notion of progress measures that was previously used in [17] to show that the powerset game, restricted to a selection of the moves of the existential player, is still sound and complete.

We also contribute with a proof of concept, a tool called `LCSFE` (Local Checker for Systems of Fixpoint Equations), which we use to show applications of the theory we present. In particular we show how our framework behaves in the settings of parity games and of the μ-calculus. We designed the tool with performance in mind, even if we do not aim to compete with the state of the art in the respective fields (e.g. PGSolver, mCRL2).

The rest of this thesis is organized as follows:

Chapter 2 Contains the theoretical background needed to understand Chapter 3. We also introduce the μ-calculus and parity games, which will be the source of examples for the tool we have developed.

Chapter 3 Here we introduce the powerset game and a way to speed up the computation of the winner of the game. This chapter uses contributions by [2]: namely selections and symbolic $\exists$-moves, and a proof that the powerset game with moves restricted to a selection is still sound and complete.

Chapter 4 We describe a local algorithm for computing the winner of the powerset game from a given position, with some details of implementation choices.

Chapter 5 We describe the tool we have created, `LCSFE`, as well as some implementation choices and issues we have encountered during development.

Chapter 6 We summarise the main contributions of this thesis and what could follow.

2
Theoretical background

In this chapter, we introduce the main theoretical tools we use throughout the whole thesis. We will focus on lattices, the computation of fixpoints and on tuples and their ordering. We also provide a brief overview on parity games.

2.1 Lattices and partial orders

Lattices are the main building block of this thesis: in particular, the domain and codomain of all the functions in a system of fixpoint equations is a (complete) lattice.

Definition 1 (preorder, poset)**.** *A partial order is relation on a set D, such that for each $x, y, z \in D$, it holds that:*

- *if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$ (Antisymmetry),*
- *$x \sqsubseteq x$ (Reflexivity),*
- *if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (Transitivity).*

If we drop the antisymmetry requirement we obtain a relation called preorder. A set D endowed with a preorder or partial order relation is usually denoted with $(D, \sqsubseteq)$; we call this type of structure preordered set and partially ordered set (or poset), respectively. From now on all sets will be paired with some kind of order relation, thus from now on we use D instead of $(D, \sqsubseteq)$.

We say that $x \in D$ is the least (respectively greatest) element of D if for all $y \in D$, $x \sqsubseteq y$ (respectively, $y \sqsubseteq x$). If such x exists, it is unique.

Definition 2 (meet and join)**.** *Given a subset $X \in D$, $d \in D$ is a least upper bound, or join of X if it is the smallest upper bound of X (that is to say, the smallest $d \in D$ such* *that for all $x \in X$, we have $x \sqsubseteq d$). We denote such element $\bigsqcup X$. The greatest of the lower bounds of X, $\bigsqcap X$, is the greatest lower bound or meet.*

Definition 3 (lattice, complete lattice)**.** *A lattice is a non-empty poset in which every pair of elements has a join and a meet. The greatest and least element of a lattice L (if they exist) are denoted with $\top_L$ and $\bot_L$, respectively. A complete lattice is a lattice $(L, \sqsubseteq)$, such that each $Y \sqsubseteq L$ has a join $\bigsqcup Y$. A complete lattice L always has a least and greatest element, and they are $\bot_L = \bigsqcup \emptyset$ and as $\top_L = \bigsqcap \emptyset$, respectively.*

Lemma 1 (finite lattice is a complete lattice)**.** *Let L be lattice with a finite number of elements, then L is a complete lattice.*

In this thesis we are going to devise a theory over complete lattices, and later, we are going to restrict ourselves to finite lattices. We are able to use the theoretical framework we are going to introduce thanks to the lemma above.

The next definition is a type of complete lattice, and we are going to use it extensively in the theoretical framework we introduce later.

Definition 4 (powerset)**.** *Given a poset X, its powerset, denoted 2^X is the set of all possible subsets of X.*

Definition 5 (basis)**.** *Given a lattice L, a basis is a subset $B_L \subseteq L$, such that for all $l \in L$, $l = \bigsqcup \{ b \in B_L \mid b \sqsubseteq l \}$.*

The intuition is that a basis B_L contains a set of elements that enables all elements of L to be built, by means of the join operator.

Example 1 (basis of powerset)**.** *Given a poset X, a basis of the powerset 2^X, is the set of singletons $B_{2^X} = \{ \{ x \} \mid x \in X \}$.*

Definition 6 (upward-closure, upward-closed set)**.** *Given a lattice L and $l \in L$, we define the upward-closure of l as: $\uparrow l = \{ l' \in L \mid l \sqsubseteq l' \}$. Given a poset X and a subset $U \sqsubseteq X$, U is an upward-closed set if, for all $x, y \in X$, such that $x \in U$ and $x \sqsubseteq y$, it is the case that $y \in U$.*

Definition 7 (monotone function on a poset)**.** *Given a poset D and a function $f : D \to D$, f is monotone if for all $x, y \in D$ such that $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$.*

We are interested in the fixpoints of a monotone function over a complete lattice, that is, given a complete lattice L and a monotone function $f : L \to L$, all $x \in L$ such that $f(x) = x$. More in detail, we want to find either the join or meet of the set of fixpoints, $\{ x \mid f(x) = x \}$.

A useful and well-known result is that in the setting of a monotone function over a complete lattice there always exist both a least and greatest fixpoint, we formalize this in the following definition.

Definition 8 (monotone function on complete lattice, least and greatest fixpoints)**.** *Given a complete lattice L, let $f : L \to L$ be a monotonic function. The least fixpoint of f, μf, is the smallest element of the set of fixpoints of f, while the greatest fixpoint, νf, is defined dually. Thanks to the Knaster-Tarski theorem [21], we know that any monotone function f on a complete lattice has a least and a greatest fixpoint, and they can be defined as, respectively:*

- $\mu f = \bigsqcap\{\, l \mid f(l) \sqsubseteq l \,\}$*, that is to say, the meet of all pre-fixpoints,*
- $\nu f = \bigsqcup\{\, l \mid l \sqsubseteq f(l) \,\}$*, the join of all post-fixpoints.*

We also have a tool to compute least and greatest fixpoint, which is the Kleene's iteration [4].

Given a complete lattice L, define its height λ_L as the supremum of the length of any strictly ascending, possibly transfinite, chain. Then we have the following result.

Theorem 1 (Kleene's iteration)**.** *Let L be a complete lattice and let $f : L \to L$ be a monotone function. Consider the (transfinite) ascending chain $(f^{\beta}(\bot))_{\beta}$, where β ranges over the ordinals, defined by $f^0(\bot) = \bot$, $f^{\alpha+1}(\bot) = f(f^{\alpha}(\bot))$ for any ordinal α and $f^{\alpha}(\bot) = \bigsqcup_{\beta \leq \alpha} f^{\beta}(\bot)$ for any limit ordinal α. Then $\mu f = f^{\gamma}(\bot)$ for some ordinal $\gamma \leq \lambda_L$. The greatest fixpoint νf can be characterised dually, via the (transfinite) descending chain $(f^{\alpha}(\top))_{\alpha}$.*

In order properly to understand this definition above we would need the notion of ordinal, and (transfinite) chain, which is outside the purpose of this thesis, since we are only concerned about finite-height complete lattices. The notion that we discern from this definition is that, given a monotone function f over a finite-height complete lattice L, we have the guarantee that by applying f iteratively, we will, eventually, reach the least or greatest fixpoint.

2.2 Tuples

Given a set A, an n-tuple belonging to set A^n will be denoted by a boldface small letter $\boldsymbol{a}$. We introduce the following notation: let $n \in \mathbb{N}$, then $\underline{n} = \{\, 1, \dots n \,\}$. Whenever we want to refer to the i-th element of a tuple $\boldsymbol{A} \in A^n$, we write a_i, with $i \in \underline{n}$. Given an n-tuple $\boldsymbol{a} \in A^n$, $i, j \in \underline{n}$ with $i \leq j$, we write $\boldsymbol{a}_{i,j}$ for the tuple $(a_i, a_{i+1}, \dots, a_j) \in A^{j-i+1}$. The empty tuple is denoted by $()$. Given two tuples $\boldsymbol{a} \in A^m$ and $\boldsymbol{a}' \in A^n$ we denote by $\boldsymbol{a}\boldsymbol{a}'$ their concatenation in A^{m+n}. We also allow concatenation between tuples and elements of a set: for example, $\boldsymbol{a}a\boldsymbol{a}' \in A^{m+n+1}$ is the concatenation between $\boldsymbol{a} \in A^m$, $a \in A$ and $\boldsymbol{a}' \in A^n$.

Definition 9 (pointwise order). *Given a poset L, $(L^n, \sqsubseteq)$ is the set of n-tuples endowed with the pointwise order defined, for $\boldsymbol{l}$, $\boldsymbol{l'} \in L^n$, by $\boldsymbol{l} \sqsubseteq \boldsymbol{l'}$ if $l_i \sqsubseteq l'_i$ for all $i \in \underline{n}$.*

2.3 Systems of fixpoint equations

Definition 10 (system of fixpoint equations). *A system of fixpoint equations E over complete lattice L is a list of m equations of the form:*

$$\begin{cases} x_1 =_{\eta_1} f_1(x_1, ..., x_m) \\ \dots \\ x_m =_{\eta_m} f_m(x_1, ..., x_m). \end{cases}$$

The empty system is $E = \emptyset$.

For all $i \in \underline{m}$, $f_i : L^m \to L$ are monotone functions, and $\eta_i \in \{\mu, \nu\}$, where ν is the greatest fixpoint, and μ is the least fixpoint.

We can describe a system of fixpoint equations E also using a tuple-like notation: $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$, where:

- $\boldsymbol{x} = (x_1, \dots, x_m)$;
- $\boldsymbol{f} = (f_1, \dots, f_m)$, which can also be seen as a function of type $\boldsymbol{f} : L^m \to L^m$;
- in $=_{\boldsymbol{\eta}}$, $\boldsymbol{\eta}$ is the tuple $(\eta_1, \dots, \eta_m)$.

Since each f_i is a monotone function, $\boldsymbol{f}$ is therefore monotone over $(L^m, \sqsubseteq)$. This allows us to characterise the solution of E as a fixpoint of the function $\boldsymbol{f} : L^m \to L^m$: the input of type L^m is the input of each function f_i, $(x_1, \dots, x_m)$, the output is a tuple $\boldsymbol{x} \in L^m$.

Definition 11 (substitution for systems of fixpoint equations). *Let E be a system of m fixpoint equations over a complete lattice L, $i \in \underline{m}$ and $l \in L$: $E[x_i := l]$ is the system of $m - 1$ equations obtained from E by removing the i-th equation and setting $x_i = l$ in all equations.*

Definition 12 (computation of the solution for systems of fixpoint equations). *The solution of E, denoted as $s = sol(E)$, $s \in L^m$ is defined inductively as:*

$$\begin{aligned} sol(\emptyset) &= () \\ sol(E) &= (sol(E[x_m := s_m]), s_m), \end{aligned}$$

where $s_m = \eta_m(\lambda x. f_m(sol(E[x_m := x]), x))$, and $()$ is an empty tuple.

We stress that the solution of a system of fixpoint equations is a tuple, thus: $sol(E[x_m := s_m]) \in L^{m-1}$, $(sol(E[x_m := s_m]), s_m) \in L^m$, and $sol_i(E)$ is the i-th component of the solution.

We recall a couple of examples from [16], in which we see the procedure we can devise from the definition above, as well as the fact that changing the order of the equations of the system, may lead to different solutions.

Example 2 (solving a system of fixpoint equations)**.** *We will now consider a system of fixpoint equations of size $m = 2$, and we will compute its solution. Consider the following system of equations E on a powerset lattice 2^A, where set A is left unspecified:*

$$E = \begin{cases} x_1 =_\mu x_1 \cap x_2 \\ x_2 =_\nu x_1 \cup x_2. \end{cases}$$

Let $f_1(x_1, x_2) = x_1 \cap x_2$, and $f_2(x_1, x_2) = x_1 \cup x_2$. First, we create the call stack, until we reach a base case:

1. *$sol(E) = (sol(E[x_2 := s_2]), s_2)$, with $s_2 = \nu(\lambda x.f_2(sol(E[x_2 := x]), x))$, let us define $E' = E[x_2 := x]$;*
2. *$sol(E') = (sol(E'[x_1 := s_1]), s_1)$, with $s_1 = \mu(\lambda x.f_1(sol(E'[x_1 := x]), x, x))$*
3. *$sol(E'[x_1 := s_1]) = ()$ and $sol(E'[x_1 := x]) = ()$.*

Going backwards we obtain that $s_1 = \mu(\lambda x.x \cup x) = A$, thus $sol(E') = ((), A) = (A)$. By substituting what we obtained in expression $\nu(\lambda x.f_2(sol(E[x_2 := x]), x))$, we have $s_2 = \nu(\lambda x.A \cap x) = A$. Finally, $sol(E) = (sol(E[x_2 := A]), A)$, and since $sol(E[x_2 := A])$ is a system with only one equation, we solve it the usual way. The result is $sol(E) = (\emptyset, A)$, since we want the least x_1.

Example 3 (order of the equations matters)**.** *We define a system of fixpoint equations, on the usual powerset 2^A, and we use it to show that the order of the equations matters, in the sense that it may change the solution.*

$$\begin{cases} x_1 =_\mu x_1 \cap x_2 \\ x_2 =_\nu x_1 \cup x_2. \end{cases}$$

In the previous example we computed the following solution, where $x_1 = \emptyset$, and $x_2 = A$. We can see that if we switch the order of the equations, the solution does not change, in fact $x_2 =_\mu \emptyset \cup x_2$ amounts to $x_2 = A$. With a slightly different system of equations, we obtain different results. Consider the following system:

$$\begin{cases} x_1 =_\mu x_1 \cup x_2 \\ x_2 =_\nu x_1 \cap x_2. \end{cases}$$

The solution of this system is the following: $x_1 = \emptyset$*, and* $x_2 = \emptyset$*. By switching the equations we obtain a different result: by executing the usual recursive algorithm we first get* $x_2 = \nu(\lambda x.x \cap x) = A$*, and by substitution,* $x_1 = A$*.*

2.4 A brief introduction to μ-calculus

The μ-calculus was conceived by Dana Scott and Jaco de Bakker, it was further developed and improved by Dexter Kozen [14] into its more popular modern version. Nowadays, it is mainly used in the description of transition systems' properties and for the verification of such properties.

First let us define the $PVar$ set of propositional variables, ranged over by x, y, z and $Prop$ of propositional symbols, ranged over by p, q, r. Whenever we exhaust the variables, we use indices, for example: $x_1 \in PVar$, $p_1 \in Prop$. We define the syntax as follows:

$$\varphi ::= \boldsymbol{t} \mid \boldsymbol{f} \mid p \mid x \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \eta x.\varphi$$

where $p \in Prop$, $x \in PVar$ and $\eta \in \{\mu, \nu\}$. Formulas of the kind $\eta x.\varphi$ are called fixpoint formulas.

The semantics of a formula is given with respect to an unlabelled transitions system $(\mathbb{S}, \to)$ where $\mathbb{S}$ is the set of states and $\to \subseteq \mathbb{S} \times \mathbb{S}$ is the transition relation. Given a formula φ and an environment $\rho : Prop \bigcup PVar \to 2^{\mathbb{S}}$ mapping each proposition or propositional variable to the set of states where it holds, we denote by $[\![\varphi]\!]_\rho \in 2^{\mathbb{S}}$ the semantics of φ defined as usual:

$$\begin{aligned}
[\![\boldsymbol{t}]\!]_\rho &= \mathbb{S} \\
[\![\boldsymbol{f}]\!]_\rho &= \emptyset \\
[\![p]\!]_\rho &= \rho(p) \\
[\![x \,\|_\rho &= \rho(x) \\
[\![\varphi_1 \wedge \varphi_2]\!]_\rho &= [\![\varphi_1]\!]_\rho \cap [\![\varphi_2]\!]_\rho \\
[\![\varphi_1 \vee \varphi_2]\!]_\rho &= [\![\varphi_1]\!]_\rho \cup [\![\varphi_2]\!]_\rho \\
[\![\Box\varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \forall t \in \mathbb{S}, s \to t \implies t \in [\![\varphi]\!]_\rho \,\} \\
[\![\Diamond\varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \exists t \in \mathbb{S}, s \to t \wedge t \in [\![\varphi]\!]_\rho \,\} \\
[\![\nu x.\varphi]\!]_\rho &= \bigcup \{\, S \subseteq \mathbb{S} \mid S \subseteq [\![\varphi]\!]_{\rho[x:=S]} \,\} \\
[\![\mu x.\varphi]\!]_\rho &= \bigcap \{\, S \subseteq \mathbb{S} \mid [\![\varphi]\!]_{\rho[x:=S]} \subseteq S \,\}.
\end{aligned}$$

Operators $\Box$ and $\Diamond$ represent the fact that, for all, and respectively some, one step evolutions of a system from a state $s \in \mathbb{S}$, φ is satisfied. Fixpoint operators μ and ν contribute to the expressiveness of the μ-calculus by allowing us to express, respectively, that "something good eventually happens" and "nothing bad ever happens", that is, liveness and safety properties. We list some examples of liveness properties:

- $\mu X.\varphi \vee \Diamond X$, that is, there is an evolution of the system where eventually φ is true;
- a property which expresses the fact that we will *eventually* satisfy φ for all possible evolutions of the system, $\text{Even}(\varphi) = \mu X.(\varphi \vee (\Box X \wedge \Diamond \boldsymbol{t}))$;
- the until operator, which expresses the fact that given two properties φ, ψ, φ is true until, eventually, ψ becomes true. We use the following notation: $\varphi \,\mathcal{U}\, \psi = \mu X.\psi \vee (\varphi \wedge \Box X \wedge \Diamond \boldsymbol{t})$.

In the following we show examples of safety properties.

- We want to verify whether φ is true for all states of the system: $\nu X.\varphi \wedge \Box X$.
- We want to verify whether there is a finite or infinite evolution of the system, where φ is always satisfied: $\text{Safe}(\varphi) = \nu X.\varphi \wedge (\Diamond X \vee \Box \boldsymbol{f})$.
- we want to express the property of a system being deadlock free, that is, the system can always make a transition. We define this property as $\varphi = \nu X.\Diamond tt \wedge \Box X$.

A generic formula $\eta x.\varphi$ can be translated to a fixpoint equation, which, we show through an example.

Example 4 (μ-calculus formula to system of fixpoint equations)**.** *Let φ be the following formula:*

$$\varphi = \nu x_2.((\mu x_1.(p \vee \Diamond x_1)) \wedge \Box x_2)$$

requiring that from all reachable states there exists a path that eventually reaches a state where p holds. The same formula can be conveniently represented via a system of fixpoint equations:

$$\begin{cases} x_1 =_\mu p \vee \Diamond x_1 \\ x_2 =_\nu x_1 \wedge \Box x_2. \end{cases}$$

2.4.1 μ-calculus on labelled transition systems

We also give the syntax and semantics of the μ-calculus over a labelled transition system. Let as define the sets of propositional variables $PVar$, propositional symbols $Prop$, and a set of labels, or actions, Act, ranged over by a. The syntax is the following:

$$\varphi ::= \boldsymbol{t} \mid \boldsymbol{f} \mid p \mid x \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [a]\varphi \mid \langle a \rangle \varphi \mid \eta x.\varphi.$$

The semantics is given with respect to a labelled transition system $(\mathbb{S}, Act, \rightarrow)$, where the semantics of the transition relation is $\rightarrow \subseteq \mathbb{S} \times Act \times \mathbb{S}$. Given $s, s' \in \mathbb{S}$, $a \in Act$, whenever we have a transition relation $(s, a, s') \in \rightarrow$, we usually write $s \xrightarrow{a} s'$. The semantics of the syntactic categories is the same except for the new modal operators:

$$\begin{aligned} [\![[a]\varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \forall t \in \mathbb{S}, a \in Act \text{ s.t. } s \xrightarrow{a} t \implies t \in [\![\varphi]\!]_\rho \,\} \\ [\![\langle a \rangle \varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \exists t \in \mathbb{S}, a \in Act \text{ s.t. } s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_\rho \,\}. \end{aligned}$$

Observation 1. *We could enrich the syntax and semantics of the μ-calculus over labelled transition systems to define modal operators that behave similarly to $\Box$ and $\Diamond$. To do this we add a new syntactic category, and respective semantics for the set of labels Act, where $a \in Act$:*

$$A := a \mid true \mid \neg a$$

$$\begin{aligned} [\![a]\!] &= \{\, a \,\} \\ [\![true]\!] &= Act \\ [\![\neg a]\!] &= Act \setminus \{\, a \,\}. \end{aligned}$$

Instead of $\langle a\rangle\varphi$ and $[a]\varphi$ we use $\langle A\rangle\varphi$ and $[A]\varphi$, with the new semantics:

$$\begin{aligned}
[\![[A]\varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \forall t \in \mathbb{S}, a \in [\![A]\!] \ s.t.\ s \xrightarrow{a} t \implies t \in [\![\varphi]\!]_\rho \,\}\\
[\![\langle A\rangle\varphi]\!]_\rho &= \{\, s \in \mathbb{S} \mid \exists t \in \mathbb{S}, a \in [\![A]\!] \ s.t.\ s \xrightarrow{a} t \land t \in [\![\varphi]\!]_\rho \,\}.
\end{aligned}$$

Formulas $\langle true\rangle\varphi$ and $[true]\varphi$, have the same meaning on labelled transition systems that $\Diamond\varphi$ and $\Box\varphi$ have on an unlabelled transition system, in the sense that they check all possible transitions from a state.

The syntax and semantics defined above are inspired from the one used in the tool mCRL2 [12].

2.5 Parity games

We borrow the definition of parity games from [13]. In the following definition we use the notation: $[d] = \{\, 0, 1, ..., d-1 \,\}$, where $d \in \mathbb{N}$.

Definition 13 (parity graph, parity game)**.** *A parity graph $G = (V, E, p)$ consists of a directed graph (V, E) and a priority function $p : V \to [d]$. A parity game $\Gamma = (V, E, p, (V_\Diamond, V_\Box))$ consists of a parity graph $G = (V, E, p)$, called the game graph of Γ, and of a partition $(V_\Diamond, V_\Box)$ of the set of vertices V.*

In a parity game there are two players $\Diamond$, $\Box$, or 0 and 1, or $\exists$ and $\forall$, respectively. We will adopt the latter notation in the next chapters. A player, usually player $\Diamond$, starts by placing a token on a vertex; the token is moved along the edges in the following way: if the token is on a vertex $v \in V_\Diamond$, then player $\Diamond$ can make a move by placing the token on a vertex adjacent to v, otherwise $\Box$ can make a move.

We define a tuple $\phi = (v_1, v_2, \ldots)$ as a play, which may or may not be finite, where $v_i \in V$ and the order of play is given by i.

Winning conditions change accordingly to the finiteness of the play:

- if the play is finite, the player that is unable to move loses;
- in an infinite play, ϕ is a winning play for $\Diamond$ ($\Box$) if the minimum priority that appears infinitely often is even (odd), this value is denoted by $\min(\mathrm{Inf}(\phi))$.

If an infinite game is being played, we could use a different winning condition, that is whether $\max(\mathrm{Inf}(\phi))$ is even or odd. We will use the latter winning condition throughout this thesis.

A game which is played with the former winning conditions can be easily converted to a game with the latter winning conditions by changing the sign of the priorities or by

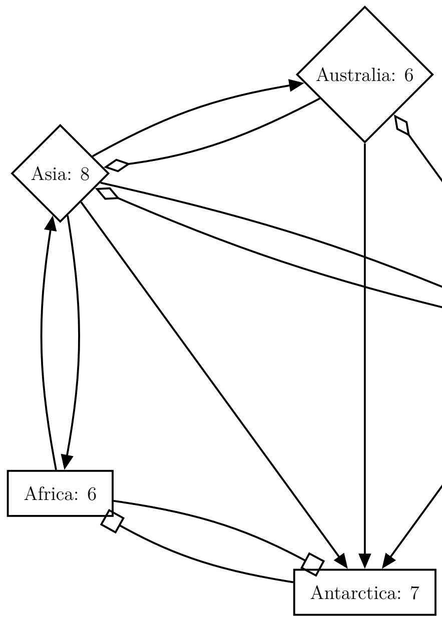

Figure 2.1: An example of a parity game. It is a modified version of an example described in the documentation of PGSolver. Diamond shaped nodes represent player 0, box shaped nodes represent player 1.

mapping the greatest priority to the least priority, the second greatest priority to the second least priority, and so on. By doing so we obtain two equivalent games that will yield the same solution.

Definition 14 (strategy, winning strategy, winning set)**.** *A strategy for player $\Diamond$ is a function $\sigma : V_\Diamond \to V$. A play $\phi = (v_1, v_2, ...)$ is said to be consistent with a strategy σ if $\sigma(v_i) = v_{i+1}$, for all $i \in \mathbb{N}$ such that $v_i \in V_\Diamond$. Dually for player $\Box$.*

A winning set, or region, for player P is a subset of vertices $W \subseteq V$, such that P can win if a play starts from $v \in W$.

A winning strategy for a player is a function σ, if for every play starting from any $v \in W$, consistent with σ, the play is winning for a player.

From the definition above we can describe the problem of solving, or deciding, a parity game: it consists in deciding whether a player has a winning strategy from the starting vertex.

Definition 15 (odd and even cycle)**.** *An odd (even) cycle is a cycle in the parity graph where the node with the smallest (greatest) priority is odd (even).*

Any parity game admits a translation into a system of fixpoint equations over the boolean lattice, that is, the lattice $\{false, true\}$, with ordering $false \sqsubseteq true$. We show this through an example.

Example 5 (example of parity game)**.** *Figure 2.1 represents a parity game: each node is paired with a continent that identifies the node and a number that represents the node's priority. We recall that player* 0 *wins either if player* 1 *cannot make a move or if there is an infinite play where the greatest priority that appears infinitely often is even, dually for player* 1*.*

Diamond-shaped vertices are owned by player 0*, while box-shaped vertices are owned by player* 1*. We represent the winning strategy of player* 0 *by using diamond-shaped arrow heads, and square arrow heads for player* 1*. For example, the strategy for player one is a function with the following mappings: Antarctica* $\mapsto$ *Africa, and Africa* $\mapsto$ *Antarctica.*

The solution of the example above was checked with the tool PGSolver, [10]. Every parity game can be converted into a system of boolean fixpoint equations via the following procedure:

1. introduce a variable for each vertex;
2. if the left-hand variable of the equation has an even (respectively odd) index, then such equation should be a greatest (respectively least) fixpoint equation;
3. the right-hand side of an equation should include all the variables that represent the successors of the node related to the left-hand variable;
4. as regards the boolean operators between variables, we should first decide whether we are interested in player 0's or in player 1's final win. The left hand side should be put under an $\bigvee$ operator if the owner of the right-hand side variable is the same as the desired winner, $\bigwedge$ otherwise;
5. the equations must be ordered by priority of the node corresponding to the left-hand variable, from smallest to greatest.

Example 6 (parity game to system of boolean fixpoint equations)**.** *We use the parity game represented in Figure 2.1, and we convert it to a system of boolean fixpoint equations. For the sake of simplicity, we relate each vertex to a number:*

- *Africa to* 0*,*
- *America to* 1*,*
- *Asia to* 2*,*
- *Australia to* 3*,*
- *Antarctica to* 4*.*

The winning region of player 0 *is given by the vertices corresponding to identifiers* $1, 2, 3$*, while player 1 is* $0, 4$*. We build the system of boolean fixpoint equations, applying the* *algorithm described above, with the goal of determining the winning region of player* 0*:*

$$\begin{cases} x_1 & =_\mu x_2 \wedge x_3 \\ x_0 & =_\nu x_2 \wedge x_4 \\ x_3 & =_\nu x_2 \vee x_4 \\ x_4 & =_\nu x_0 \\ x_2 & =_\mu x_0 \vee x_1 \vee x_3 \vee x_4. \end{cases}$$

The solution of this system of boolean fixpoint equations is $x_1 = true$, $x_2 = true$, $x_3 = true$: this is because nodes $1, 2, 3$ are in the winning region of player 0, and $x_4 = false$, $x_0 = false$, because nodes 4, 0 are not. By substituting the values in the system, we can verify that the solution of the parity game is consistent with to the system of equations.

3

Selections and symbolic ∃-moves

3.1 Powerset game

In this section we introduce the powerset game: we characterise the solution of a system of fixpoint equations over a complete lattice in terms of a parity game.

Definition 16 (powerset game)**.** *Let L be a complete lattice and let B_L be a basis for L. Let E be a system of m fixpoint equations over L, of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$.*

The powerset game associated with E is a parity game with two players:

- *$\exists$, the existential player, with positions (b, i);*
- *$\forall$, the universal player, with positions $\boldsymbol{X} = (X_1, \dots, X_m) \in (2^{B_L})^m$;*
- *from position (b, i) the possible moves of $\exists$ are $\boldsymbol{E}(b, i) = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq f_i(\bigsqcup \boldsymbol{X}) \,\}$;*
- *from position $\boldsymbol{X} \in (2^{B_L})^m$ the possible moves of $\forall$ are $\boldsymbol{A}(\boldsymbol{X}) = \{\, (b, i) \mid i \in \underline{m} \wedge b \in X_i \,\}$.*

In a finite play the winner is the player whose opponent is unable to move. In an infinite play the condition is the following: let h be the highest index that occurs infinitely often in a pair (position) (b, i):

- *if $\eta_h = \nu$ then $\exists$ wins,*
- *if $\eta_h = \mu$ then $\forall$ wins.*

Remark 1 (basis and $\bot$)**.** *Note that $\bot$ of L is not necessarily included in B_L, because for $l = \bot$ we have that $\bot = \bigsqcup \emptyset$ (by definition of lattice). This is consistent with the*

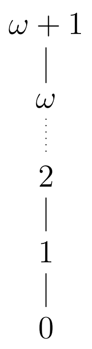

Figure 3.1: An example of infinite lattice.

definition of basis, which is a set such that $\forall l \in L$, $l = \bigsqcup\{b \in B_L \mid b \sqsubseteq l\}$, *when there is no* $b \in B_L$ *such that* $b \sqsubseteq l$, *we have* $\emptyset$. *Moreover, if* $\forall$ *played* $(\bot, i)$ *for some* $i \in \underline{m}$ *player* $\exists$ *could answer with a tuple* $(\emptyset, \ldots, \emptyset) \in (2^{B_L})^m$, *and win the game.*

This game theoretical characterisation opens the way to the development of algorithms to find or verify solutions of systems of fixpoint equations: given a complete lattice and a basis, the game allows us to verify whether an element b is *covered* by the solution, in the sense that it is smaller than the solution, with respect to some ordering.

Theorem 2 (correctness and completeness of powerset game)**.** *Let* E *be a system of* m *fixpoint equations over a complete lattice* L, *with solution* $\boldsymbol{s}$. *For all* $b \in B_L$, *and* $i \in \underline{m}$, *we have that* $b \sqsubseteq s_i$ *if and only if player* $\exists$ *has a winning strategy from position* (b, i).

The proof of this lemma can be found in [2].

Example 7 (game on an infinite lattice)**.** *Consider the lattice* $L = \mathbb{N} \cup \{\omega, \omega+1\}$ *represented by the Hasse diagram in Figure 3.1 and let* $B_L = L$. *We define* $f : L \to L$ *as a monotone function with* $f(n) = n+1$ *for* $n \in \mathbb{N}$ *and* $f(\omega) = \omega$, $f(\omega+1) = \omega+1$. *Consider the powerset game played on a system composed by only* f, *of which we want the least fixpoint, clearly* $\mu f = \omega$. *Now suppose that we want to verify whether* $\omega \leq \mu f$. *On an infinite play clearly* $\forall$ *wins, since we are playing a game on a single* μ*-equation. The play starts from* $b = \omega$. *An example of an infinite play is a one where* $\exists$ *answers to the starting move with* $X = \{\,\omega+1\,\}$, *to which* $\forall$ *answers with* $\omega+1$, *which clearly unfolds into an infinite play. In a winning play for* $\exists$ *she answers with a move* $X = \mathbb{N}$, *this is because* $b = \omega \leq f(\bigsqcup \mathbb{N}) = f(\omega)$. *Player* $\forall$ *can answer with any* $n \in \mathbb{N}$. *From now on,* $\exists$ *can always play* $X = \{n-1\}$, *which leads to a descending chain where player* $\exists$ *wins by finally playing* $X = \emptyset$, *to which* $\forall$ *cannot answer.*

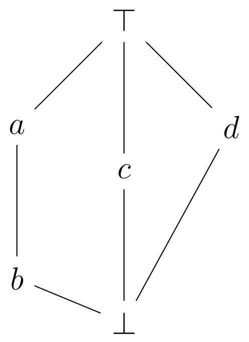

Figure 3.2: Hasse diagram of a finite lattice L.

3.2 Selections

Let $\boldsymbol{s}$ be the solution of a system of fixpoint equations. Thanks to Theorem 2 we can answer the question of whether if $b \sqsubseteq s_i$, with $i \in \underline{m}$. We would have to compute a subset of player $\exists$'s moves, if not the whole set, which is unreasonable, since $\exists$ could play a huge number of moves. We can restrict the set of moves of the existential player without affecting the correctness and completeness of the game. We do so by showing that some moves are useless; by removing them we create a subset of moves called selection.

Let us first introduce an ordering relation on sets, called Hoare preorder [1].

Definition 17 (Hoare preorder)**.** *Given a poset P, consider the so-called Hoare preorder, $\sqsubseteq_H$ on P, defined by letting, for $X, Y \subseteq P$, $X \sqsubseteq_H Y$ if for all $x \in X$ there exists a $y \in Y$ such that $x \sqsubseteq y$.*

Remark 2 (Hoare preorder is not anti-symmetric)**.** *The Hoare preorder is not anti-symmetric in general, take for example the lattice L in Figure 3.2 and its powerset 2^L: we have that $\{ a, b, c \} \sqsubseteq_H \{ a, c \} \sqsubseteq_H \{ a, b, c \}$, but the two sets are not equal.*

We use the following intuition. Let $\boldsymbol{X}, \boldsymbol{Y} \in \boldsymbol{E}(b, i)$: since both $\boldsymbol{X}, \boldsymbol{Y}$ are tuples in $(2^{B_L})^m$ we can order them by using the pointwise extension of $\sqsubseteq_H$, $\sqsubseteq_H^\wedge$. We say that $\boldsymbol{X} \sqsubseteq_H^\wedge \boldsymbol{Y}$ when $\boldsymbol{X}_i \sqsubseteq_H \boldsymbol{Y}_i$, for all $i \in \underline{m}$. If $\boldsymbol{X} \sqsubseteq_H^\wedge \boldsymbol{Y}$, we can avoid playing $\boldsymbol{Y}$, without affecting the winner of the game: player $\exists$ must descend as much as possible to win; playing larger elements would be inconvenient from its point of view.

Definition 18 (selection)**.** *Let L be a lattice. A selection is a function $\sigma : (B_L \times \underline{m}) \to 2^{(2^{B_L})^m}$ such that for all (b, i) it holds that $\uparrow_{\mathrm{H}} \sigma(b, i) = \boldsymbol{E}(b, i)$.*

Where $\uparrow_{\mathrm{H}} \sigma(b, i) = \{ \boldsymbol{X} \in (2^{B_L})^m \mid \exists \boldsymbol{Y} \in \sigma(b, i) \wedge \boldsymbol{Y} \sqsubseteq_H^\wedge \boldsymbol{X} \}$, and $\uparrow_{\mathrm{H}}$ is the upward-closure with respect to $\sqsubseteq_H^\wedge$.

Requiring set $\boldsymbol{E}(b, i)$ to be the upward-closure of $\sigma(b, i)$ with respect to $\sqsubseteq_{\mathrm{H}}^\wedge$ is equivalent to requiring that $\sigma(b,i) \subseteq \boldsymbol{E}(b,i)$ and for each $\boldsymbol{X} \in \boldsymbol{E}(b,i)$ there exists $\boldsymbol{Y} \in \sigma(b,i)$ such that $\boldsymbol{Y} \sqsubseteq_{\mathrm{H}}^{\wedge} \boldsymbol{X}$.

Lemma 2 (Hoare preorder preserves supremum)**.** *Let X, Y be two sets in 2^{B_L}. If $X \sqsubseteq_H Y$ then $\bigsqcup X \sqsubseteq \bigsqcup Y$.*

Proof. Observe that, since $X \sqsubseteq_H Y$, there exists $y \in Y$ so that $x \sqsubseteq y$ for all $x \in X$, then it must be the case that $\bigsqcup X \sqsubseteq \bigsqcup Y$.

□

Theorem 3 ($\boldsymbol{E}(b,i)$ is upward-closed with respect to $\sqsubseteq_{\mathrm{H}}^{\wedge}$)**.** *Let L be a complete lattice, and B_L be one of its bases. Let E be a system of $m \in \mathbb{N}$ fixpoint equations $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. For each $b \in B_L$, $i \in \underline{m}$, the set of moves of $\exists$ is upward-closed with respect to $\sqsubseteq_{\mathrm{H}}^{\wedge}$, meaning $\boldsymbol{E}(b,i) = \uparrow_{\mathrm{H}} \boldsymbol{E}(b,i)$.*

Proof. Let us remember that $\boldsymbol{E}(b,i) = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq f_i(\bigsqcup \boldsymbol{X}) \,\}$. To prove that $\boldsymbol{E}(b,i)$ is upward-closed we need to show that, given $\boldsymbol{X}, \boldsymbol{Y} \in (2^{B_L})^m$ so that $\boldsymbol{X} \sqsubseteq_{\mathrm{H}}^{\wedge} \boldsymbol{Y}$, if $\boldsymbol{X} \in \boldsymbol{E}(b,i)$ it must be the case that $\boldsymbol{Y} \in \boldsymbol{E}(b,i)$. If $\boldsymbol{X} \in \boldsymbol{E}(b,i)$, by definition we have that $b \sqsubseteq f_i(\bigsqcup \boldsymbol{X})$. By definition of $\sqsubseteq_{\mathrm{H}}^{\wedge}$ we have that for all $i \in \underline{m}$, $X_i \sqsubseteq_H Y_i$, thus by Lemma 2, we obtain that $\bigsqcup X_i \sqsubseteq \bigsqcup Y_i$, then it must be the case that $\bigsqcup \boldsymbol{X} \sqsubseteq \bigsqcup \boldsymbol{Y}$. Thanks to the monotonicity of f_i we have that $b \sqsubseteq f_i(\bigsqcup \boldsymbol{X}) \sqsubseteq f_i(\bigsqcup \boldsymbol{Y})$, then by definition of $\boldsymbol{E}(b,i)$, $\boldsymbol{Y} \in \boldsymbol{E}(b,i)$.

□

Theorem 4 (game with selection)**.** *Let E be a system of m equations over a complete lattice L of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ with solution $\boldsymbol{s}$, and let σ be a selection for E. For all $b \in B_L$ and $i \in \underline{m}$, $b \sqsubseteq s_i$ if, and only if, $\exists$ has a winning strategy from position (b,i) in the game restricted to the selection σ.*

Proof. This proof is taken from [2]. We first prove correctness, or $(\Leftarrow)$. If $b \not\sqsubseteq s_i$ then by Theorem 2 we know that in the original game, $\forall$ would have a winning strategy from position (b,i). It is the same for the game restricted to the selection, since they do not affect $\forall$ moves.

We now prove completeness. Let $b \sqsubseteq s_i$, so that by Theorem 2 player $\exists$ has a winning strategy t from position (b,i), in the original game. Our goal is to show that $\exists$ has a winning strategy in the restricted game, starting from every position (b',i) such that $b' \sqsubseteq b$, hence also from b itself, independently of the moves of $\forall$. In the original game, according to winning strategy t, from position (b,i) player $\exists$ would play $\boldsymbol{X} = t(b,i)$. Since $b' \sqsubseteq b$, we know that $\boldsymbol{X} \in \boldsymbol{E}(b,i) \subseteq \boldsymbol{E}(b',i)$, by definition. Thus, by definition of selection there must exist $\boldsymbol{Y} \in \sigma(b',i)$ such that $\boldsymbol{Y} \sqsubseteq_{\mathrm{H}}^{\wedge} \boldsymbol{X}$. Now, we have two cases depending on

$\boldsymbol{X}$. If all components of $\boldsymbol{X}$ are empty, then it must be the case for $\boldsymbol{Y}$ as well: player $\forall$ has no possible moves and $\exists$ wins. Otherwise, let (b'', j) be every move $\forall$ can play from position $\boldsymbol{Y}$. It is the case that for all $b'' \in Y_j \sqsubseteq_H X_j$, there must exist some $b''' \in X_j$, such that $b'' \sqsubseteq b'''$. In the original game this means that $(b''', j) \in \boldsymbol{A}(t(b, i) = \boldsymbol{X})$, and since $\exists$ wins playing strategy t, the latter must be a winning strategy from position (b''', j) as well. We can apply the same reasoning to (b''', j), and we could go on indefinitely, until either $\exists$ wins, or we obtain a pair of infinite plays, with the following shape:

$$\begin{aligned}
&(b_0, i_0) \to \boldsymbol{X_0} \to (b_1, i_1) \to \boldsymbol{X_1} \to (b_2, i_2) \to \boldsymbol{X_2} \to \dots \\
&(b'_0, i_0) \to \boldsymbol{Y_0} \to (b'_1, i_1) \to \boldsymbol{Y_1} \to (b'_2, i_2) \to \boldsymbol{Y_2} \to \dots
\end{aligned}$$

where $(b_0, i_0) = (b, i)$ and $b'_1 = b'$, and for all k we have $b'_k \sqsubseteq b_k$ and $\boldsymbol{X}^k = t(b_k, i_k) \sqsupseteq_H^{\wedge} \boldsymbol{Y}^k \in \sigma(b'_k, i_k)$. The first play above is a play in the original game, where $\exists$ wins. Since the second play is from the game restricted to a selection σ, and it has the same indices i_k, we have that $\exists$ wins from the restricted game as well.

$\square$

3.3 Least selection

We showed that player $\exists$ can win a version of the powerset game restricted to a selection. We know that $\sigma(b, i) \subseteq \boldsymbol{E}(b, i)$, and clearly, for computational purposes, we want $\sigma(b, i)$ to be as small as possible. To do that we need to introduce an ordering relation on selections.

Definition 19 (order on selections)**.** *Let E be a system of m equations of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$, over a complete lattice L, with basis B_L. Given two selections σ, σ', we write $\sigma \subseteq_H \sigma'$ if for all $b \in B_L$, $i \in \underline{m}$, and $\boldsymbol{X} \in \sigma(b, i)$, there exists $\boldsymbol{Y} \in \sigma'(b, i)$ such that $\boldsymbol{X} \subseteq^{\wedge} \boldsymbol{Y}$.*

Unfortunately, the existence of a minimal selection with respect to $\subseteq_H$ is not always guaranteed. We show this through an example.

Example 8 (minimal and least selection on infinite height lattices)**.** *Consider the game described in Example 7. We recall that it is a game played on the lattice $L = \mathbb{N} \cup \{ \omega, \omega+1 \}$, with basis $B_L = L$, on a single μ-equation defined as follows*

$$f(n) = \begin{cases} n+1 & n \in \mathbb{N} \\ n & \textit{if } n = \omega \textit{ or } n = \omega + 1. \end{cases}$$

Player $\exists$ wins a game from the starting position (ω) by forcing the play to be finite. In Example 7 we observed that $\exists$ can win by playing $X = \mathbb{N}$ initially and then by answering

$X = \{\, n-1 \,\}$ whenever player $\forall$ plays n. Player $\exists$ can also win by always playing the set $\{\, m \in \mathbb{N} \mid m \le n-1 \,\}$. We show that this both kinds of moves correspond to a selection, and that they are not minimal with respect to $\subseteq_H$.

$$\sigma(n) = \begin{cases} \{\, \{\, n-1 \,\} \,\} & \textit{if } n \in \mathbb{N} \\ \{\, \mathbb{N} \,\} & \textit{if } n = \omega \\ \{\, \{\, \omega+1 \,\} \,\} & \textit{if } n = \omega+1 \end{cases}$$

$$\sigma'(n) = \begin{cases} \{\, \{\, m \in \mathbb{N} \mid m \le n-1 \,\} \,\} & \textit{if } n \in \mathbb{N} \\ \{\, \mathbb{N} \,\} & \textit{if } n = \omega \\ \{\, \{\, \omega+1 \,\} \,\} & \textit{if } n = \omega+1 \end{cases}$$

Since we have only one equation in the system, we omit the indices and thus the tuple notation is no longer necessary: the selection $\sigma : B_L \times \underline{m} \to 2^{(2^{B_L})^m}$ becomes $\sigma : B_L \to 2^{2^{B_L}}$.

Both σ and σ' are valid selections, i.e., $\uparrow_{\mathrm{H}} \sigma(b) = \uparrow_{\mathrm{H}} \sigma'(b) = E(b)$ for all $b \in B_L$. Moreover, the two selections differ if $n \in \mathbb{N}$, and the unique $X \in \sigma(n)$ is always included in the unique $X' \in \sigma'(n)$, hence $\sigma \subseteq_H \sigma'$. Notably, σ is not minimal since there exist smaller selections, infinitely many in fact. For instance, take σ'' defined as σ except for $\sigma''(\omega) = 2\mathbb{N}$ the set of even numbers. Clearly σ'' is still a valid selection, $\sigma'' \subseteq_H \sigma$ and clearly there are even smaller ones (but no minimal).

From now on we will consider only finite-height complete lattices: by doing this we can prove the existence of a selection which is minimal and unique, hence it is the least selection.

Theorem 5 (least selection)**.** *Let E be a system of m equations over a complete lattice L with finite height. Then, there exists a unique selection σ such that $\sigma \subseteq_H \sigma'$ for all selections σ'.*

Proof. For the proof of the existence of a least selection for finite-height complete lattices, we refer the reader to [2]. □

3.4 A logic for upward-closed sets

From Theorem 3, we know that the set of moves of the existential player is upward-closed. Upward-closed sets can be conveniently represented and manipulated in logical form (see, e.g., [5]). Intuitively, (minimal) selections describe a disjunctive normal form, but more compact representations can be obtained using arbitrary nesting of conjunction and disjunction.

Example 9 (minimal selections are exponential in size)**.** *For instance, the minimal selection for the monotone function $f(X_1, \dots, X_{2n}) = (X_1 \cup X_2) \cap (X_3 \cup X_4) \cap \dots \cap (X_{2n-1} \cup X_{2n})$ would be of exponential size (think of the corresponding disjunctive normal form). Suppose that L is a generic complete lattice and $b \in B_L$, and that f is the i-th equation of a system of fixpoint equations $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. The minimal selection of f over b would have the following shape:*

$$\sigma(b,i) = \{\, (\{\, b \,\}, \emptyset, \{\, b \,\}, \emptyset, \dots, \{\, b \,\}, \emptyset), \dots, (\emptyset, \{\, b \,\}, \{\, b \,\}, \emptyset, \dots, \{\, b \,\}, \emptyset) \,\}.$$

In this example $\sigma(b,i)$ has $2^{n/2}$ tuples, which represent all possible combinations of ways we can cover all the disjunctions in the minimal selection.

This motivates the introduction of a propositional logic for expressing the set of moves of the existential player in a linear size formula along with a technique for extracting moves from such logic formulas.

Definition 20 (logic for upward-closed sets, with respect to $\sqsubseteq_{\mathrm{H}}^{\wedge}$)**.** *Let L be a lattice and let B_L be a basis for L. Given $m \in \mathbb{N}$, the logic $L_m^H(B_L)$ has the following inductive definition, where $b \in B_L$ and $j \in \underline{m}$:*

$$\varphi ::= [b,j] | \bigwedge_{k \in K} \varphi_k | \bigvee_{k \in K} \varphi_k.$$

We *true* the empty conjunction for $\bigwedge_{k\in\emptyset} \varphi_k$, and $false$ for the empty disjunction $\bigvee_{k\in\emptyset} \varphi_k$.

Whenever $\varphi = [b,j]$, *true* or $false$, we refer to it just as atom, because in functions that work inductively over logic formulas, whenever the input is $[b,j]$, *true* or $false$, we do not need any recursive step to compute the function solution.

The connection between this language and upward-closed sets is attained via the following semantics.

Definition 21 (semantics for logic formulas)**.** *The semantics of a formula φ is an upward-closed set $[\![\varphi]\!] \subseteq (2^{B_L})^m$ with respect to $\sqsubseteq_{\mathrm{H}}^{\wedge}$, it is defined as follows:*

$$\begin{aligned}
[\![[b,j]]\!] &= \{\, \boldsymbol{X} \in (2^{B_L})^m \mid \{\, b \,\} \sqsubseteq_H X_j \,\} \\
[\![\bigwedge_{k\in K} \varphi_k]\!] &= \bigcap_{k \in K} [\![\varphi_k]\!] \\
[\![\bigvee_{k\in K} \varphi_k]\!] &= \bigcup_{k \in K} [\![\varphi_k]\!].
\end{aligned}$$

We also give the semantics for $true$ and $false$ atoms:

$$
\begin{aligned}
[\![true]\!] &= [\![\bigwedge_{k\in\emptyset} \varphi_k]\!] = \bigcap_{k\in\emptyset} [\![\varphi_k]\!] = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \,\} \\
[\![false]\!] &= [\![\bigvee_{k\in\emptyset} \varphi_k]\!] = \bigcup_{k\in\emptyset} [\![\varphi_k]\!] = \emptyset.
\end{aligned}
$$

The definition of the semantics for logic formulas serves a dual purpose: the first is to ensure that all formulas represent upward-closed sets, the second is to have a connection between formulas and selections. We show the latter in the following example, which we took from [16].

Example 10 (semantic for logic formulas)**.** *Let us define the semantics of an atom* $[b, i]$ *in a different way*

$$
[\![[b, j]]\!]_s = \{\, \boldsymbol{X} \in (2^{B_L})^m \mid b \sqsubseteq \bigsqcup X_j \,\}. \tag{3.1}
$$

This definition yields formulas with a lower number of atoms, compared to the semantics in Definition 21. But by using Definition 3.1, the connection with the least selection is not as immediate as it is when using the other one.

We show this in a concrete example. Consider the lattice L *in Figure 3.2, with the basis* $B_L = \{\, a, b, c, d \,\}$*. Suppose we have a system of equations* E *with a single equation, such that*

$$
\begin{aligned}
\boldsymbol{E}(b, 1) &= \{\, X \subseteq B_L \mid b \sqsubseteq \bigsqcup X \,\} \\
&= \{\, (\{\, b \,\}), (\{\, a, b \,\}), (\{\, b, c \,\}), (\{\, b, d \,\}), \\
&\quad\; (\{\, a \,\}), (\{\, a, c \,\}), (\{\, a, d \,\}), (\{\, c, d \,\}), \\
&\quad\; (\{\, b, c, d \,\}), (\{\, a, c, d \,\}), (\{\, a, b, c, d \,\}) \,\}.
\end{aligned}
$$

The corresponding selection is $\sigma(b, 1) = \{\, (\{\, b \,\}), (\{\, c, d \,\}) \,\}$ *since* $\uparrow_{\mathrm{H}}\{\, (\{\, b \,\}), (\{\, c, d \,\}) \,\} = \boldsymbol{E}(b, 1)$*. Let* φ_e *be the formula for the semantics in Definition 21, and* φ_s *for the semantics we defined last. Both formulas represent the same least selection* σ*, but they differ in the way they do it:*

$$
\begin{aligned}
\varphi_e &= [b, 1] \vee ([c, 1] \wedge [d, 1]) \\
\varphi_s &= [b, 1].
\end{aligned}
$$

Intuitively, φ_e is going to be a disjunction, where each disjunct represents a move in the selection. Each move is going to be a conjunction between the atoms stemming from each b in such move. However, this is not always the case. Consider Example 9, where we see that the minimal selection is of exponential size, while the symbolic $\exists$-move would be of linear size.

Intuitively, the meaning of an atom $[b, j]$ in our semantics is an upward-closed set, therefore the whole semantics yields upward-closed sets. The next lemma states that every upward-closed set can be represented by a formula in the language we just defined.

Lemma 3 (formulas for upward-closed set, with respect to $\sqsubseteq^{\wedge}_{\mathrm{H}}$)**.** *Let L be a lattice with basis B_L and let $X \subseteq (2^{B_L})^m$ be an upward-closed set with respect to $\sqsubseteq^{\wedge}_{\mathrm{H}}$. Then $X = [\![\varphi]\!]$ where φ is the formula in $\mathcal{L}^H_m(B_L)$ defined as follows:*

$$\varphi = \bigvee_{\boldsymbol{Y} \in X} \bigwedge \{\, [b, j] \mid j \in \underline{m} \wedge \{\, b \,\} \sqsubseteq_H Y_j \,\}$$

Proof. We have to show that $X = [\![\varphi]\!]$.

- ($\supseteq$) Let $\boldsymbol{Y'} \in [\![\varphi]\!]$: by applying the semantics we obtain:

  $$\boldsymbol{Y'} \in \bigcup_{\boldsymbol{Y} \in X} \bigcap \{\, \{\, \boldsymbol{Y''} \in (2^{B_L})^m \mid \{\, b \,\} \sqsubseteq_H Y''_k \,\} \mid k \in \underline{m} \wedge \{\, b \,\} \sqsubseteq_H Y_k \,\}.$$

  Hence there exists $\boldsymbol{Y} \in X$ such that for all $j \in \underline{m}$ and $\{\, b \,\} \sqsubseteq_H Y_j$ it holds that $\{\, b \,\} \sqsubseteq_H Y'_j$. Then

  $$Y_j = \{\, b \mid b \in B_L \wedge \{\, b \,\} \sqsubseteq_H Y_j \,\} \sqsubseteq_H \{\, b \mid b \in B_L \wedge \{\, b \,\} \sqsubseteq_H Y'_j \,\} = Y'_j$$

  for all $i \in \underline{m}$. Hence $\boldsymbol{Y} \sqsubseteq^{\wedge}_{\mathrm{H}} \boldsymbol{Y'}$, and we have that $\boldsymbol{Y'} \in X$, since X is upward-closed.

- ($\subseteq$) Let $\boldsymbol{Y} \in X$. We show that $\boldsymbol{Y} \in [\![\psi_{\boldsymbol{Y}}]\!]$, where $\psi_{\boldsymbol{Y}} = \bigwedge\{\, [b, j] \mid j \in \underline{m} \wedge \{\, b \,\} \sqsubseteq_H Y_j \,\}$. In fact,

  $$[\![\psi_{\boldsymbol{Y}}]\!] = \bigcap \{\, \{\, \boldsymbol{Y'} \in (2^{B_L})^m \mid \{\, b \,\} \sqsubseteq_H Y'_j \,\} \mid j \in \underline{m} \wedge \{\, b \,\} \sqsubseteq_H Y_j \,\}.$$

  Now, if $j \in \underline{m}$ and $\{\, b \,\} \sqsubseteq_H Y_j$, then clearly $\boldsymbol{Y} \in \{\, \boldsymbol{Y'} \mid \{\, b \,\} \sqsubseteq_H Y'_j \,\}$, and hence $\boldsymbol{Y}$ is contained in the intersection.

□

For practical purposes we should restrict to finite formulas. This can surely be done in the case of finite lattices, and so from now on we will work only with finite lattices.

Definition 22 (symbolic $\exists$-moves)**.** *Let L be a finite lattice and let $f : L^m \to L$ be a monotone function. A symbolic $\exists$-move for f is a family $(\varphi_b)_{b \in B_L}$ of formulas in $\mathcal{L}^H_m(B_L)$ such that, for all $b \in B_L$ it holds that $[\![\varphi_b]\!] = \boldsymbol{E}(b, f)$, where $\boldsymbol{E}(b, f) = \{\, \boldsymbol{X} \in (2^{B_L})^m \mid b \sqsubseteq f(\bigsqcup \boldsymbol{X}) \,\}$. If E is a system of m equations of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ over L, a symbolic $\exists$-move for E is a family of formulas $(\varphi^i_b)_{b \in B_L, i \in \underline{m}}$ such that for all $i \in \underline{m}$, the family $(\varphi^i_b)_{b \in B_L}$ is a symbolic $\exists$-move for f_i.*

We can compose symbolic $\exists$-moves. This has a useful implication: if we have a formula for all operators used in a system of fixpoint equations (e.g., the formulas for the modal operators $\Box$ and $\Diamond$ of the μ-calculus), we can easily build, by composition, the symbolic $\exists$-moves for the system.

Theorem 6 (composition of symbolic $\exists$-moves)**.** *Let L be a complete lattice with a basis B_L, and let $f : L^n \to L$, $f_j : L^m \to L$ for $j \in \underline{n}$ be monotone functions and let $(\varphi_b)_{b \in B_L}$, $(\varphi^j_b)_{b \in B_L, j \in \underline{n}}$ be symbolic $\exists$-moves for $f, f_1, \ldots, f_n$. Consider the function $h : L^m \to L$ obtained as the composition $h(\boldsymbol{x}) = f(f_1(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x}))$. Define $(\varphi'_b)_{b \in B_L}$ as follows. For all $b \in B_L$, the formula $(\varphi'_b)_{b \in B_L}$ is obtained from φ_b by replacing each occurrence of $[b', j]$ by $\varphi^j_{b'}$. Then $(\varphi'_b)_{b \in B_L}$ is a symbolic $\exists$-move for h.*

Proof. We first show that given a formula $\varphi \in \mathcal{L}^H_n(B_L)$, if φ' is the formula in $\mathcal{L}^H_m(B_L)$ obtained from φ by replacing each occurrence of an atom $[b, j]$ by φ^j_b, then

$$[\![\varphi']\!] = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![\varphi^j_b]\!] \,\}.$$

We proceed by induction on φ_b.

- ($\varphi = [b, j]$): In this case $\varphi' = \varphi^j_b$. We have

$$\begin{aligned} [\![\varphi']\!] &= [\![\varphi^b_j]\!] \\ &= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge \{\, b \,\} \sqsubseteq_H \{\, f_j(\bigsqcup \boldsymbol{X}) \,\} \,\} \\ &= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![[b, j]]\!] \,\} \\ &= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![\varphi]\!] \,\}. \end{aligned}$$

- ($\varphi = \bigvee_{k \in K} \varphi_k$): We have $\varphi' = \bigvee_{k \in K} \varphi'_k$, where each φ'_k is obtained from φ_k by replacing each occurrence of an atom $[b, j]$ by φ^j_b. Then

$$\begin{aligned}
[\![\varphi']\!] &= [\![\bigvee_{k\in K} \varphi'_k]\!] \\
&= \bigcup_{k\in K} [\![\varphi'_k]\!] \\
&= \bigcup_{k\in K} \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\} \in [\![\varphi_k]\!]) \,\} \quad \text{[by inductive hyp.]} \\
&= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in \bigcup_{k\in K} [\![\varphi_k]\!] \,\} \\
&= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![\bigvee_{k\in K} \varphi_k]\!] \,\} \\
&= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![\varphi]\!] \,\}.
\end{aligned}$$

- $(\varphi = \bigwedge_{k\in K} \varphi_k)$: Analogous.

Now, given $b \in B_L$ we must show that

$$\begin{aligned}
[\![\varphi'_b]\!] = \boldsymbol{E}(b,h) &= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq h(\bigsqcup \boldsymbol{X}) \,\} \\
&= \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq f(f_1(\bigsqcup \boldsymbol{X}), \ldots, f_n(\bigsqcup \boldsymbol{X})) \,\}
\end{aligned}$$

which follows from what we proved above, in fact:

$$\begin{aligned}
&[\![\varphi'_b]\!] = \\
&\quad = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge (\{\, f_1(\bigsqcup \boldsymbol{X}) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\}) \in [\![\varphi_b]\!] \,\} && (3.2) \\
&\quad = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq f(\bigsqcup(\{\, f_1(\bigsqcup \boldsymbol{X})) \,\}, \ldots, \{\, f_n(\bigsqcup \boldsymbol{X}) \,\})) \,\} && (3.3) \\
&\quad = \{\, \boldsymbol{X} \mid \boldsymbol{X} \in (2^{B_L})^m \wedge b \sqsubseteq f(f_1(\bigsqcup \boldsymbol{X}), \ldots, f_n(\bigsqcup \boldsymbol{X})) \,\}. && (3.4)
\end{aligned}$$

Equation 3.2 is true by the property proved above. By definition of symbolic ∃-move we obtain Equation 3.3 and finally by applying the meet operator we reach Equation 3.4, which concludes the proof.

□

The next definition gives us an algorithm to extract moves for the existential player from the symbolic representation; moreover it connects selections with symbolic ∃-moves. This algorithm comes as a function called $M^{\exists}$. Intuitively, our goal is, given a symbolic ∃-move $\varphi^i_b \in \mathcal{L}^H_m(B_L)$ for a function f_i, to compute a set of moves for the existential player, such that it is the smallest set of moves and is as close as possible to selection $\sigma(b,i)$.

Definition 23 (generator for symbolic $\exists$-moves)**.** *Let E be a system of m fixpoint equations over a finite lattice L, of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. The generator $M^{\exists} : \mathcal{L}_m^H(B_L) \to 2^{(2^{B_L})^m}$ is defined as follows, for $b \in B_L$ and $j \in \underline{m}$:*

$$
\begin{aligned}
M^{\exists}([b,j]) &= \{\ \boldsymbol{X}\ \} \text{ such that } X_j = \{\, b \,\} \wedge X_i = \emptyset \text{ for } i \neq j \\
M^{\exists}(\bigwedge_{k \in K} \varphi_k) &= \bigcup \{\, \cup^{\wedge} \{\, \boldsymbol{X} \mid \boldsymbol{X} \in y \,\} \mid y \in \prod_{k \in K} M^{\exists}(\varphi_k) \,\} \\
M^{\exists}(\bigvee_{k \in K} \varphi_k) &= \bigcup \{\, M^{\exists}(\varphi_k) \mid k \in K \,\}
\end{aligned}
$$

The definition above connects this theoretical framework with the local algorithm described in Chapter 4.

The next lemma formalizes the fact that using the generator $M^{\exists}$ in the powerset game, instead of a selection, leads to a game that is correct and complete.

Lemma 4 (symbolic $\exists$-moves and least selection)**.** *Let E be a system of m fixpoint equations over a finite lattice L of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. Let $(\varphi_b^i)_{b \in B_L, i \in \underline{m}}$ be a symbolic $\exists$-move for E. Given the least selection $\hat{\sigma}$ for E, for every position $(b,i) \in (B_L \times m)$, it holds that*

$$\hat{\sigma}(b,i) \subseteq M^{\exists}(\varphi_b^i) \subseteq \uparrow_{\mathrm{H}} \hat{\sigma}(b,i) = \boldsymbol{E}(b,i).$$

We only give an idea of the proof of the lemma above: after showing that $\hat{\sigma}(b,i) \subseteq M^{\exists}([b,i]) \subseteq \uparrow_{\mathrm{H}} \hat{\sigma}(b,i)$ the rest follows by simple structural induction on the shape of the formula.

4
Local algorithm

We present an algorithm to verify local solutions for systems of fixpoint equations. Thanks to Theorem 2, we can leverage the notion of powerset game so that our problem becomes to verify whether there exists a winning strategy for player $\exists$ from the starting node. More in detail, the algorithm finds whether $b \sqsubseteq s_i$, where $\boldsymbol{s}$ is the solution of a system of fixpoint equations $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ on a complete lattice L and $b \in B_L$ is an element of the basis of the lattice.

The algorithm arises as a generalization of the one in [20]. The idea is that we explore the game graph of the powerset game, storing the path walked and gathering assumptions and decisions about the nodes we visit. Whenever we establish that a player can win from a node, we backtrack, propagating the information backwards. The algorithm stops whenever we find a winning strategy from the initial position of player $\exists$ or exhaust all her possible moves, resulting in player $\forall$ winning. The original local algorithm is from [2], in this thesis we also build on a contribution by [17]: we use symbolic $\exists$-moves to efficiently store and generate new moves for the existential player. Moreover, we introduce in the pseudocode notable improvements from the tool described in Chapter 5.

4.1 Notation

For the rest of the chapter, L denotes a fixed complete lattice, with basis B_L, and E a system of m fixpoint equations over L, of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$, with solution $\boldsymbol{s} \in L^m$. A generic player is denoted by P, and its opponent by $\overline{P}$. The set of all positions of the game is denoted by $Pos = Pos_\exists \cup Pos_\forall$, where $Pos_\exists = B_L \times \underline{m}$, ranged over by (b, i) is the set of positions controlled by $\exists$, and $Pos_\forall = (2^{B_L})^m$, ranged over by $\boldsymbol{X}$, is the set of positions controlled by $\forall$. A generic position is denoted by uppercase C, and $P(C)$ is the player controlling position C. Given a position $C = (b, i) \in Pos_\exists$, we denote a symbolic $\exists$-move φ_b^i by φ_C. We extract moves for player $\forall$ using a function $M^\forall : Pos_\forall \times \mathbb{N} \to Pos_\exists \times \mathbb{N} \times Pos_\forall$, whose pseudocode is presented in Algorithm 1: given a position of $\forall$ we extract a b from $C = \boldsymbol{X} \in Pos_\forall$ and we return a position (b, i) along with a number j which is the index of the X_j from which we will extract the next move and an updated $\boldsymbol{X}$ without the b we have just chosen. Clearly, whenever we return a value $j > m$, we have exhausted all possible moves for $\forall$ from $\boldsymbol{X}$.

Algorithm 1: The function $M^\forall$

```
Require: There is at least a move for $\forall$, meaning, there must be $b \in X_i$ for some
         $i \in \underline{m}$.

function $M^\forall(\boldsymbol{X}, i)$:
    while $X_i = \emptyset$ do
        $i \leftarrow i + 1$
    pick $b$ from $X_i$
    $X \leftarrow X_i \setminus \{ b \}$
    $\boldsymbol{X}' \leftarrow \boldsymbol{X}_{1,i-1} X \boldsymbol{X}_{i+1,m}$
    $j \leftarrow i$
    while $j > m \vee X = \emptyset$ do
        $j \leftarrow j + 1$
    return $(b, i), j, \boldsymbol{X}'$
```

We extract moves in this manner for $\forall$ to make the process *lazy* and we are going to use a similar idea for $\exists$. In previous versions of the local algorithm, the whole set of moves is generated whenever we needed the set possible moves from a position: it is reasonable to think that there are instances where we do not actually need to compute the whole set of moves for a player, since we do not need to visit the whole game graph to find a winning strategy from a node. We define a function $\mathrm{I} : Pos \to (\underline{m} \cup \{ 0 \})$, which maps every position to a priority, which is i for (b, i) and 0 for any $\boldsymbol{X} \in Pos_\forall$. We suppose we have the set of symbolic $\exists$-moves for E available, and that we can retrieve a formula for the position $C = (b, i)$, at any time. When we refer to a formula for a generic position $C = (b, i)$ we will write φ_C, instead of φ_b^i. With this notation the winning condition can be expressed as follows:

- a finite play is won by the player who moved last;
- an infinite play $(C_1, C_2, \ldots)$, is won by player $\exists$ (resp. $\forall$) if there exists a priority $h \in \underline{m}$ such that $\eta_h = \nu$ (resp. μ), and the set $\{ j \mid \mathrm{I}(C_j) = h \}$ is infinite and the set $\{ j \mid \mathrm{I}(C_j) > h \}$ is finite.

Note that the largest index which occurs infinitely often cannot be 0 since only positions of player $\forall$ have priority 0 and players alternate during the game. Hence, the highest priority in a play will always be a priority computed from a position $(b, i) \in Pos_\exists$.

4.2 The algorithm

The procedure roughly consists in a depth-first exploration of the tree of plays arising as unfolding of the game graph starting from initial position (b, i). The algorithm optimises the search by making assumptions on particular subtrees, which are thus pruned. Assumptions can be later confirmed or invalidated and thus withdrawn.

The algorithm is split in two main functions:

- function Explore, which explores the tree of plays of the game, trying different moves from each node to determine the player who has a winning strategy from such node;
- function Backtrack, which backtracks from a node after the algorithm has established who wins from it and propagates this information backwards.

Moreover, there are six auxiliary functions:

- function Forget, which removes erroneous decisions made during the search;
- functions NextMove and BuildNextMove, that generate a move for player $\exists$ from a formula;
- functions Reduce, ApplyDecisions and Unfold, which update a formula φ to keep $\exists$ from playing unnecessary moves.

4.2.1 Data structures

The algorithm uses the following data structures.

- *Counter* $\boldsymbol{k} \in \mathbb{N}^m$, an m-tuple of natural numbers that associates each non-zero priority with the number of times the priority has been encountered in the play since a higher priority was last faced (the current position is not included). After each move, the counter is updated considering the priority of the current position. More precisely, the update of counter $\boldsymbol{k}$, moving from a position with priority i, denoted $\text{Next}(\boldsymbol{k}, i)$, is defined as follows: $\text{Next}(\boldsymbol{k}, i)_j = 0$, for all $j < i$, $\text{Next}(\boldsymbol{k}, i)_i = k_i + 1$ and $\text{Next}(\boldsymbol{k}, i)_j = k_j$, for all $j > i$. We also define a total order on counters $<_P$, that measures how good the current advancement of the game is for player P. We let $\boldsymbol{k} <_\exists \boldsymbol{k}'$ when the largest i such that $k_i \neq k'_i$ is the index of a greatest fixpoint equation and $k_i < k'_i$, or it is the index of a least fixpoint and $k_i > k'_i$. We say that $\boldsymbol{k} <_\forall \boldsymbol{k}'$ if, and only if $\boldsymbol{k}' <_\exists \boldsymbol{k}$. We write $\boldsymbol{k} \leq_P \boldsymbol{k}'$ for $\boldsymbol{k} <_P \boldsymbol{k}'$ or $\boldsymbol{k} = \boldsymbol{k}'$.

- *Playlist* ρ, a list of the positions encountered from the root to the current node (empty if the current node is the root), each with counter $\boldsymbol{k}$, and π, which is a tuple that represents the alternative moves not yet explored. Thus ρ is a list of triples $(C, \boldsymbol{k}, \pi)$, where C is a position, $\boldsymbol{k}$ a counter corresponding to the move that led to position C and π is a tuple such that:
  - if C is a move for the existential player and $\boldsymbol{k}$ is the corresponding counter, π is a tuple $(\varphi, \boldsymbol{k})$, where φ is a formula that represents unexplored moves for $\exists$ from C;
  - if C is a move for the universal player and $\boldsymbol{k}$ is the corresponding counter, π is a tuple with three elements $(\boldsymbol{X}, j, \boldsymbol{k})$, where $\boldsymbol{X}$ stores the remaining alternative moves for $\forall$ and $j \in \mathbb{N}$ represents the j-th element of $\boldsymbol{X}$, X_j, from which we are going to extract the next alternative move.
- *Assumptions* for player $\exists$ and $\forall$, that is, a tuple of sets $\Gamma = (\Gamma_\exists, \Gamma_\forall)$. We store assumptions for each player. A position C is assumed to be winning for some player when it is encountered for the second time in the current playlist ρ. This reveals the presence of a loop in the game graph, which can be unfolded into an infinite play. A position C is assumed to be winning for the player that would win in such infinite play. This ensures that the highest priority in the loop is the index of a least fixpoint if $P = \forall$ and of a greatest fixpoint if $P = \exists$. The assumption is stored with the corresponding counter: Γ_P contains pairs of the kind $(C, \boldsymbol{k})$. Since other possible paths branching from the loop are possibly unexplored, assumptions can be proved false during the exploration.
- *Decisions* for player $\exists$ and $\forall$, that is, a pair of finite sets $\Delta = (\Delta_\exists, \Delta_\forall)$. Intuitively, a decision for a player P is a position C of the game from which we established that P has a winning strategy. The decision is stored with the corresponding counter. The result of this is that Δ_P contains pairs of the kind $(C, \boldsymbol{k})$.

We suppose that each time we add either a new assumption or a new decision, we store a *timestamp* along with the new tuple. We do not show this in the pseudocode.

4.2.2 Generation of moves for the existential player

Moves for the existential player are stored in a formula φ. Intuitively, we can extract moves from φ by using an algorithm based on Definition 23. Moreover, we can use the set of decisions Γ and the playlist ρ to reduce the set of moves available to $\exists$ from a position. Intuitively, we just anticipate what could have happened if player $\exists$ chose these moves.

If we could somehow make the whole process *lazy*, we would have a solution similar to that we used for the universal player. But this is not necessary, since we can modify and store the formula directly, so that whenever we want a move for $\exists$ we can simply extract one from the current φ. We can modify the formula by using the decisions and the playlist we currently have, making sure that we do not visit the same position more than once if we already have information about it.

4.2.2.1 Reducing the formula

Given a formula φ, a counter $\boldsymbol{k}$, the set of decisions Δ and a playlist ρ, the function Reduce$(\varphi, \boldsymbol{k}, \Delta, \rho)$, in Algorithm 2, returns a triple $(\varphi', \Gamma_\exists, \Gamma_\forall)$, where φ' is the simplified symbolic $\exists$-move equipped with two sets of new assumptions, one for each player.

Algorithm 2: The function Reduce

```
function Reduce($\varphi, \boldsymbol{k}, \Delta, \rho$):
    $(\varphi', \Gamma_\exists, \Gamma_\forall) \leftarrow$ ApplyDecisions$(\varphi, \boldsymbol{k}, \Delta, \rho)$
    return Simplify$(\varphi')$, $\Gamma_\exists$, $\Gamma_\forall$
```

The function Reduce uses two auxiliary functions: ApplyDecisions and Simplify. The function ApplyDecisions, described in Algorithm 3, takes all Reduce input parameters and transforms the atoms of the given formula into *true* or *false*, using the information contained in Δ and ρ. We describe the behaviour of this function: it first performs a recursive unfolding of the formula until it reaches all the atoms by using the function Unfold. Whenever we reach an atom $[b, i]$, we check:

- whether there is a decision for the position (b, i), with a favourable $\boldsymbol{k}'$ for player P, that is, if there is a $\boldsymbol{k}'$ such that $((b, i), \boldsymbol{k}') \in \Delta_P$ and $\boldsymbol{k}' \leq_P \boldsymbol{k}$. Hence, $P = \exists$ means that the atom $[b, i]$ would lead to a win for player $\exists$ and so the value returned is *true*. Otherwise, $[b, i]$ would lead to a win for player $\forall$, so in this case the function returns *false*;
- whether position (b, i) has already been encountered in the play, i.e., if there is a $\boldsymbol{k}'$ such that $((b, i), \boldsymbol{k}', \_) \in \rho$ and $\boldsymbol{k}' <_P \boldsymbol{k}$; in this case (b, i) becomes an assumption for P. As before, if $P = \exists$ the atom becomes true, false otherwise.

Finally, the function ApplyDecisions returns a triple $(\varphi', \Gamma_\exists, \Gamma_\forall)$, where φ' is the formula with some atoms transformed into *true* or *false*; the set $\Gamma_\exists$ includes all the new assumptions favorable for $\exists$, and dually $\Gamma_\forall$ is the set of new assumptions for $\forall$.

We do not show the pseudocode for the function Simplify, but just provide an intuition. Let us define a function Simplify : $\mathcal{L}_m^H(B_L) \to \mathcal{L}_m^H(B_L)$, such that during the exploration of the tree of the input formula, certain subformulas are recursively replaced with the following:

Algorithm 3: Functions ApplyDecisions and Unfold

```
function ApplyDecisions($\varphi, \boldsymbol{k}, \Delta, \rho$):
    match $\varphi$
        case $[b, i]$
            if there is $((b, i), \boldsymbol{k}') \in \Delta_\exists$ s.t. $\boldsymbol{k}' \leq_\exists \boldsymbol{k}$ then
                return $(true, \emptyset, \emptyset)$
            else if there is $((b, i), \boldsymbol{k}') \in \Delta_\forall$ s.t. $\boldsymbol{k}' \leq_\forall \boldsymbol{k}$ then
                return $(false, \emptyset, \emptyset)$
            else if there is $((b, i), \boldsymbol{k}') \in \rho$ s.t. $\boldsymbol{k}' <_\exists \boldsymbol{k}$ then
                return $(true, \{ ((b, i), \boldsymbol{k}') \}, \emptyset)$
            else if there is $((b, i), \boldsymbol{k}') \in \rho$ s.t. $\boldsymbol{k}' <_\forall \boldsymbol{k}$ then
                return $(false, \emptyset, \{ ((b, i), \boldsymbol{k}') \})$
            else
                return $(\varphi, \emptyset, \emptyset)$
        case $\bigvee_{j \in J} \varphi_j$
            return Unfold($\varphi, J, \vee, false$)
        case $\bigwedge_{j \in J} \varphi_j$
            return Unfold($\varphi, J, \wedge, true$)
function Unfold($\varphi, J, op, init$):
    $(\Gamma_1, \Gamma_2) \leftarrow (\emptyset, \emptyset)$
    $\varphi' \leftarrow init$
    foreach $j \in J$ do
        $(\varphi', \Gamma_\exists, \Gamma_\forall) \leftarrow$ApplyDecisions($\varphi_j, \boldsymbol{k}, \Delta, \rho$)
        $(\Gamma_1, \Gamma_2) \leftarrow (\Gamma_1 \cup \Gamma_\exists, \Gamma_2 \cup \Gamma_\forall)$
        $\varphi' \leftarrow \varphi' \; op \; \varphi'_j$
        return $\varphi, \Gamma_1, \Gamma_2$
```

$$\begin{aligned}\textsc{Simplify}(x \wedge \mathit{true}) &= x\\ \textsc{Simplify}(x \wedge \mathit{false}) &= \mathit{false}\\ \textsc{Simplify}(x \vee \mathit{true}) &= \mathit{true}\\ \textsc{Simplify}(x \vee \mathit{false}) &= x\end{aligned}$$

where $x, \mathit{true}, \mathit{false}$ are formulas and operators $\wedge, \vee$ are commutative. Function SIMPLIFY receives a formula φ as input and returns φ', which is either an atom or a formula that does not contain true or false atoms.

4.2.2.2 BUILDING THE NEXT MOVE

Given a formula φ, the function NEXTMOVE(φ), which we provided in Algorithm 4, returns a move for $\exists$. We require the function to be called after a simplification phase, which removes true and false atoms from the formula, and require the formula itself to not be a false atom. All these requirements are fullfilled in EXPLORE, see Algorithm 7, where NEXTMOVE is called after REDUCE. If we simplify φ to a false atom we avoid computing the next move. The function NEXTMOVE(φ) does the following:

- if φ is a true atom, then it returns the emptyset tuple $\boldsymbol{\emptyset}$, lowest possible move we can choose among all the moves in $(2^{B_L})^m$;
- otherwise, it calls the auxiliary function BUILDNEXTMOVE.

Algorithm 4: The function NEXTMOVE

```
Require: $\varphi$ is not $false$ and it was simplified using REDUCE.
function NEXTMOVE($\varphi$):
    match $\varphi$
        case $true$
            return $\boldsymbol{\emptyset}$
        other
            return BUILDNEXTMOVE($\varphi$)
```

The function BUILDNEXTMOVE has the same requirements as NEXTMOVE. We are sure that they are met since it is always called inside NEXTMOVE. The function BUILD-NEXTMOVE unfolds the formula until it reaches an atom $[b, i]$, and it returns a tuple where all components are $\emptyset$ except for the i-th one, which is $\{ b \}$. For $\wedge$ nodes we return the union of the recursive calls applied to the subformulas, while instead for $\vee$ nodes we return the recursive call of just one subformula of the node. The “pick” statement could choose either a random or a fixed formula; it does not affect the correctness of the function, but it does affect its performance. Intuitively, there are “better” moves among the set of available moves: it is usually convenient to give priority to moves which are immediately reducible to valid decisions or assumptions for the player who is moving (see [2]). Intuitively, players could choose each turn the best possible move available from a position, which corresponds to a winning strategy. Of course, this is not possible since it would require a The pseudocode of BuildNextMove is defined in Algorithm 5.

These two algorithms use the function in Definition 23. Intuitively, each time we call NextMove(φ) we return just one move among the moves we could generate by using $M^{\exists}(\varphi)$.

Algorithm 5: The function BuildNextMove

```
Require: $\varphi$ doesn't contain $true$ or $false$ leaves.
function BuildNextMove($\varphi$):
    $C \leftarrow \emptyset$
    match $\varphi$
        case $[b, i]$
            $C_i = \{ b \}$
        case $\bigwedge_{j \in J} \varphi_j$
            foreach $j \in J$ do
                $C \leftarrow C \cup^{\wedge}$ BuildNextMove($\varphi_j$)
        case $\bigvee_{j \in J} \varphi_j$
            pick any $j \in J$
            $C \leftarrow$ BuildNextMove($\varphi_j$)
    return $C$
```

4.2.3 Explore and backtrack

We call the function Explore($(b, i), \mathbf{0}, [], (\emptyset, \emptyset), (\emptyset, \emptyset)$) for checking whether $b \sqsubseteq s_i$ for $b \in B_L$ and $i \in m$, where $\mathbf{0}$ is the everywhere-zero counter. This returns the only player P having a winning strategy from position (b, i) and, by Theorem 2, $P = \exists$ if, and only if $b \sqsubseteq s_i$. Intuitively, whenever we call Explore we walk along a path in the game graph, while saving it in the playlist ρ. The walk goes on until we find a node from which a player has a winning strategy. Then we backtrack, propagating the information backwards: while backtracking we try alternative moves for each player. This procedure terminates when, during backtracking, we go back to the starting position, i.e., the playlist ρ is empty and we do not have alternative moves to try. The player that wins from the starting position is the overall winner.

Let us now describe the function EXPLORE, whose pseudocode is provided in Algorithm 7. Given a position C, the corresponding counter $\boldsymbol{k}$, the playlist ρ, and the set of assumptions Γ and decisions Δ, the function checks whether one of the following conditions hold, and answers accordingly.

- If there are no moves from position C, the player controlling C, $P(C)$, cannot move and its opponent $\overline{P(C)}$ wins. This check is operated by the function ISEMPTY (see Algorithm 6). Therefore, a new decision for the current position C, along with a timestamp, is added for the opponent, and then we backtrack.
- If there is already a favourable decision for P for the current position C, that is, $(C, \boldsymbol{k}') \in \Delta_P$ and $\boldsymbol{k}' \leq_P \boldsymbol{k}$, we can use that information to assert that P would win from the current position as well. The requirement $\boldsymbol{k}' \leq_P \boldsymbol{k}$ intuitively ensures that we arrived at the current position C with a play that is at least as good for P as the play which led to the previous decision $(C, \boldsymbol{k}')$.
- If the current position C was already encountered in the play, meaning that there is $(C, \boldsymbol{k}, \pi) \in \rho$ for some $\boldsymbol{k}'$, such that $\boldsymbol{k}' <_P \boldsymbol{k}$, then C becomes an assumption for P, the player for which the counter became strictly better. Then we backtrack.
- If none of the conditions above holds and $P = \forall$, the exploration continues from C. A move $C' \in M^\forall(C, 1)$ is chosen to be explored. The playlist is extended by adding $(C, \boldsymbol{k}, \pi)$, where π records the remaining moves to be explored. Counter $\boldsymbol{k}$ is updated according to the priority of C.
- If none of the conditions above holds and the $P = \exists$, we first retrieve the associated formula φ_C from the current position $C \in Pos_\exists$. Then we simplify the logic formula using the function REDUCE. If any assumptions are created, we merge them pointwise to the current set of assumptions Γ, that is, we combine assumptions for each player P with the newly created assumptions for P. Now we check whether the simplified formula φ'_C became false; if so, we add C, along with its counter, as a decision for the universal player and we backtrack. Otherwise, we explore a new move C', extracted with NEXTMOVE. The playlist is extended by adding $(C, \boldsymbol{k}, \pi)$, where π stores the simplified symbolic $\exists$-move φ'_C. Counter $\boldsymbol{k}$ is updated according to the priority of the past position C.

We now describe the backtracking procedure, presented in Algorithm 8. The function BACKTRACK$(P, \rho, \Gamma, \Delta)$ is used to backtrack from a position C, from the top of playlist ρ, after assuming or deciding that player P will win from this position. The BACKTRACK algorithm works as follows. If the playlist is empty, we are back at the root, the position from which the computation started, then the exploration concludes and the algorithm decides that player P is the winner from such a position. Otherwise, we pop the head $(C', \boldsymbol{k}, \pi)$ of the playlist ρ and the status of position C' is investigated.

Algorithm 6: The function IsEmpty

```
function IsEmpty($C$):
    match $C$
        case $\boldsymbol{X}$
            return $false$ if $X_i \in \emptyset$, for all $i \in \underline{m}$, $true$ otherwise
        case $(b, i)$
            get $\varphi_C$ from position $C$
            return $\varphi_C = false$
```

Algorithm 7: The function Explore

```
function Explore($C, \boldsymbol{k}, \rho, \Gamma, \Delta$):
    if IsEmpty($C$) then
        $\Delta_{\overline{P(C)}} \leftarrow \Delta_{\overline{P(C)}} \cup \{ (C, \boldsymbol{k}) \}$
        return Backtrack($\overline{P(C)}, \rho, \Gamma, \Delta$)
    else if there is $P$ s.t. $(C, \boldsymbol{k}') \in \Delta_P \wedge \boldsymbol{k}' \leq_P \boldsymbol{k}$ then
        return Backtrack($P, \rho, \Gamma, \Delta$)
    else if there is $P$ s.t. $(C, \boldsymbol{k}', \_) \in \rho \wedge \boldsymbol{k}' <_P \boldsymbol{k}$ then
        $\Gamma_P \leftarrow \Gamma_P \cup \{ (C, \boldsymbol{k}') \}$
        return Backtrack($P, \rho, \Gamma, \Delta$)
    else
        $\boldsymbol{k}' \leftarrow$ Next($\boldsymbol{k}$,I($C$))
        match $C$
            case $\boldsymbol{X}$
                $(C', j, \boldsymbol{X}') \leftarrow M^{\forall}(\boldsymbol{X}, 1)$
                $\pi \leftarrow (\boldsymbol{X}', j, \boldsymbol{k}')$
                return Explore($C', \boldsymbol{k}', ((C, \boldsymbol{k}, \pi) :: \rho), \Gamma, \Delta$)
            case $(b, i)$
                get $\varphi_C$ from $C$
                $(\varphi'_C, \Gamma'_{\exists}, \Gamma'_{\forall}) \leftarrow$ Reduce($\varphi_C, \boldsymbol{k}', ((C, \boldsymbol{k}) :: \rho), \Delta$)
                $\Gamma \leftarrow (\Gamma_{\exists} \cup \Gamma'_{\exists}, \Gamma_{\forall} \cup \Gamma'_{\forall})$
                if $\varphi'_C = false$ then
                    $\Delta_{\forall} \leftarrow \Delta_{\forall} \cup \{ (C, \boldsymbol{k}) \}$
                    return Backtrack($\forall, \rho, \Gamma, \Delta$)
                else
                    $C' \leftarrow$ NextMove($\varphi'_C$)
                    $\pi \leftarrow (\varphi'_C, \boldsymbol{k}')$
                    return Explore($C', \boldsymbol{k}', ((C, \boldsymbol{k}, \pi) :: \rho), \Gamma, \Delta$)
```

- If C' is controlled by the opponent of P, such that $P(C') \neq P$, we face two cases:
  - if $\forall$ controls C' and there are still unexplored moves from C', that is, $\pi = (\boldsymbol{X}, j, \boldsymbol{k}'')$ and $j \leq m$, we must explore such moves before deciding the winner from C'. Then a new move is extracted and we explore the tree of plays from it;
  - if, instead, $\exists$ controls C', we retrieve $(\varphi, \boldsymbol{k}'')$ from π. Then we simplify φ to φ' and we add the newly created assumptions, if any. If $\varphi' \neq false$, we retrieve the next move C'' to be explored from φ'. Before exploring the move C'', we replace the formula φ with φ' in ρ, with the same counter $\boldsymbol{k}''$.
- At this point either $P(C') = P$ or there are no unexplored moves from C'. In both cases C' is winning for player P, hence we insert C' with the corresponding counter in Δ_P, along with a timestamp. Since we decided that P would win from C' we can now continue to backtrack. Before backtracking, however, we must discard all assumptions for the opponent of P in conflict with the newly taken decision, and this must be propagated to the decisions depending on such assumptions. This is done by the invocation $\textsc{Forget}(\Delta_{\overline{P}}, \Gamma_{\overline{P}}, (C', \boldsymbol{k}'))$.

4.2.3.1 Forget

Each time we add a new decision or assumption, we also associated a timestamp with it. From the pseudocode that we presented it is not apparent that all decisions are supported by other decisions and assumptions. We have already seen that we may prove some assumptions false during the exploration. Intuitively, when an assumption in Γ_P fails and is withdrawn, a sound Forget function must delete at least all decisions relying on the assumption that were proved false. We implemented a sound Forget function which only considers timestamps: whenever an assumption is proved false, we remove all the decisions that have been made after the assumption. This, by experimentation, proved to be a more efficient approach than removing only the set of decisions depending on the assumption which have been proven wrong, at least in the setting of the μ-calculus. Otherwise, managing decisions and assumptions becomes complex and inefficient. Different sound realizations of Forget and a more thorough analysis of this issue can be found in [20].

4.3 Correctness

Here we provide an intuition of how correctness can be proved for the local algorithm. A detailed correctness proof for the original algorithm can be found in [2]. In order to formally show that our algorithm is correct, we would need to prove the following theorem.

Theorem 7 (correctness)**.** *Given a fixpoint game, if a call* $\textsc{Explore}(C, \mathbf{0}, [], (\emptyset, \emptyset), (\emptyset, \emptyset))$ *returns a player P, then P wins the game from C.*

Algorithm 8: The function BACKTRACK

```
function BACKTRACK($P, \rho, \Gamma, \Delta$):
    if $\rho = []$ then
        return $P$
    let $C', \boldsymbol{k}', \pi, t$ s.t. $\rho = ((C', \boldsymbol{k}', \pi) :: t)$
    if $\mathrm{P}(C') \neq P$ then
        if $\mathrm{P}(C') = \forall$ then
            let $\boldsymbol{X}, j, \boldsymbol{k}''$ s.t. $\pi = (\boldsymbol{X}, j, \boldsymbol{k}'')$
            if $j \leq m$ then
                $(C'', j', \boldsymbol{X}') \leftarrow M^{\forall}(\boldsymbol{X}, j)$
                $\pi' \leftarrow (\boldsymbol{X}', j', \boldsymbol{k}'')$
                return EXPLORE($C'', \boldsymbol{k}'', ((C', \boldsymbol{k}', \pi') :: t), \Gamma, \Delta$)
        if $\mathrm{P}(C') = \exists$ then
            let $\varphi, \boldsymbol{k}''$ s.t. $\pi = (\varphi, \boldsymbol{k}'')$
            $(\varphi'_C, \Gamma'_{\exists}, \Gamma'_{\forall}) \leftarrow$ REDUCE($\varphi, \boldsymbol{k}'', \rho, \Delta$)
            $\Gamma \leftarrow (\Gamma_{\exists} \cup \Gamma'_{\exists}, \Gamma_{\forall} \cup \Gamma'_{\forall})$
            if $\varphi'_C \neq false$ then
                $C'' \leftarrow$ NEXTMOVE($\varphi'_C$)
                $\pi' \leftarrow (\varphi'_C, \boldsymbol{k}'')$
                return EXPLORE($C'', \boldsymbol{k}'', ((C', \boldsymbol{k}', \pi') :: t), \Gamma, \Delta$)
    $\Delta_P \leftarrow \Delta_P \cup \{ (C', \boldsymbol{k}') \}$
    $\Gamma_P \leftarrow \Gamma_P \setminus \{ (C', \boldsymbol{k}') \}$
    if there is $(C', \boldsymbol{k}') \in \Gamma_{\overline{P}}$ then
        $\Delta_{\overline{P}} \leftarrow$ FORGET($\Delta_{\overline{P}}, \Gamma_{\overline{P}}, (C', \boldsymbol{k}')$)
        $\Gamma_{\overline{P}} \leftarrow \Gamma_{\overline{P}} \setminus \{ (C', \boldsymbol{k}') \}$
    return BACKTRACK($P, t, \Gamma, \Delta$)
```

The proof of correctness in [2] relies on the following lemma.

Lemma 5 (termination)**.** *Given a fixpoint game on a finite lattice, any call* $\textsc{Explore}(C_0, \mathbf{0}, [], (\emptyset, \emptyset), (\emptyset, \emptyset))$ *terminates with an invocation of the function* $\textsc{Backtrack}(P, C_0, [], (\emptyset, \emptyset), \Delta)$ *for some player P and a set Δ.*

We modified the backtracking algorithm in [2] by allowing player $\exists$ to anticipate future moves among the unexplored ones. It is not convenient for $\exists$ to play moves which would be winning for player $\forall$, according to either the set of decisions and the playlist. Moreover, we make new assumptions whenever we are able to anticipate that a move would be either winning or losing by looking inside playlist ρ. As a consequence of this modification the termination lemma no longer holds. Here we discuss how we can prove our algorithm correct. A completely formal proof is needed, ad it is regarded as a possible future contribution.

The only change with respect to the algorithm in [2] is the choice of moves for the existential player. Intuitively, since we are only restricting the moves played by the existential player, and only pruning plays for which we already have information, the algorithm is still correct.

More specifically, we distinguish two possibilities concerning the moves of $\exists$ not explored in the algorithm.

- The move discarded from player $\exists$ does not belong to a selection. This case is completely covered by Theorem 4, which states that the game played with moves restricted to selections is equivalent to the original game played without any restriction.
- The move discarded from player $\exists$ belongs to a selection. Let us call this move $C \in Pos_\forall$. Since C was discarded during the simplification step, a decision or assumption $(C, \boldsymbol{k})$ must have led to the transformation of the corresponding atom to $true$ or $false$. If we had played C in the next exploration step, we would have reached the same conclusions, since $(C, \boldsymbol{k})$ would have been applied when exploring the said atom. Indeed, this arises from the fact that the same necessary conditions, specified in the function Explore, are also checked in ApplyDecisions. which can be seen by comparing Algorithm 7 and Algorithm 3.

In this version of the local algorithm, we anticipate future moves. It is important for the sake of correctness that whenever we add a new assumption it comes from the playlist ρ, and the counter we use to verify whether a move in the playlist is repeated is the counter corresponding to the move we are going to choose in the future. In function Reduce both conditions are satisfied: we look for new assumptions in ρ and player $\forall$ does not modify the counter, moves are anticipated of one step only, that is, the next move of $\forall$.

5 The tool: LCSFE

The main contribution of this thesis is a tool called `LCSFE`, (Local Checking for Systems of Fixpoint Equations). The source code of this project can be found at link https://github.com/strang3nt/LCSFE. The tool implements the algorithm described in Chapter 4. It receives as input a specification language consisting of a system fixpoint equations with monotone operators over a complete lattice and the symbolic ∃-moves for each operator. It then verifies whether an element of the lattice is covered by the solution of the system of fixpoint equations.

In particular, the user can provide customized monotone operators and their respective symbolic ∃-moves, making it possible to instantiate our tool for many types of verification logics, provided that they can be encoded into systems of fixpoint equations.

We provide, along with our tool, two modules containing examples of applications of our theoretical framework, namely, a local solver for parity games, and a model checker for the μ-calculus. Each module performs the encoding of a specification language to a system of fixpoint equations, and computes the symbolic ∃-moves for their respective operators.

Firstly, we will introduce Rust, the language we chose for this project. Then, we will make some mentions of PEGs (Parsing Expression Grammars) and parser combinators: in our project we used a parser library that uses both such techniques. Finally, we will describe the project's design choices.

5.1 A brief introduction to Rust

In this section we give a brief introduction to the programming language Rust, our main goals being to highlight some of its features and to introduce some of the specific Rust terminology that we will use in the rest of this chapter.

Rust is a relatively new programming language, but thanks to its characteristics has soon become one of the most important programming languages to date. Rust promises performance to rival the likes of C++ and C, while using a modern syntax and enforcing memory safety. Rust borrows characteristics from both functional and object-oriented programming; it has support for sum types, pattern matching, clojures, but it can also realise side effects and has support for mutable collections and advanced operations with memory. The type system is minimal: along with the usual primitive types it supports the *struct* type, the sum type (called *enum*), and the *trait*. The trait is the only way to obtain some form of inheritance. Structs, enums, traits and functions are then organized in *modules*. One or more modules working together form a *crate*. In the rest of this chapter, we will be using the term module quite loosely and usually to refer to a crate. Whenever we refer to a module in the proper sense, it will be made clear by the context.

Memory management in Rust is based on ownership: each value is *owned* by a variable, and a value is immediately deleted whenever the variable that owns it goes out of scope. Owned values can be borrowed and intuitively a borrow is a reference to a value. Moreover, there are precise rules about how owned values can be borrowed by other variables, functions, or other constructs. Ownership is enforced by the borrow checker at compile time. The borrow checker is usually able to do most of the work for the programmer, but there are situations where the programmer is required to declare how long references should be valid: in this case we are required to specify the so-called *lifetime* of a reference. This system guarantees type safety and no memory leaks, but it comes at a cost: the learning curve is quite steep.

The Rust Standard Library is minimal, but useful nonetheless. It supports a set of basic collections: vectors, lists, and maps. We can also do operations on collections via iterators, which are provided by the Rust Standard Library as well. Iterators in Rust are *lazy*, in the sense that no computation occurs until they are required to generate an element. Rust also supports a number of smart pointers, which are particularly useful when managing lifetimes becomes too hard and to allocate values on the heap. The most notable smart pointers are `Box<T>` which is a smart pointer in the line of C++ unique pointers, and `Rc<T>`, which is a type of pointer whose ownership can be shared, similarly to C++ shared pointers.

One of the reasons for Rust's success is Cargo: it is a build system, dependency and package manager and is used in the vast majority of Rust based projects. It makes compiling, running and testing Rust projects easy and seamless.

5.2 Parsing Expression Grammars and parser combinators

In `LCSFE` we made extensive use of a parsing library called Chumsky, whose source code can be found at link https://github.com/zesterer/chumsky. Chumsky is a type of parser based on parser combinators, capable of parsing PEGs. We provide a brief description of both terms and their implications.

Parser combinators are a parsing technique that consists in implementing parsers by defining them in terms of other parsers. Thus, parser combinators are merely functions accepting other parser combinators as input. A parser in this setting can be viewed as a function that accepts a string and returns some kind of data structure. This approach is quite different from parser generators (e.g., Bison, ANTLR), which are a kind of parser that accepts as input a grammar specification to then generate a source code that can parse streams of characters, using the grammar. This process, aside from writing the grammar, usually requires little to no intervention by the developer. Problems start later in the development whenever the grammar needs to be changed: both the grammar and the generated source code are going to need maintenance. The main advantage of parser combinators, and the main reason why we choose a parser based on them, is quick prototyping. A simple language can be described by a hundred lines of code, which is exactly our setting.

PEG is a type of formalism used to describe languages and protocols which is quite different from context-free grammars (CFGs). The main advantage of PEG, with respect to CFG and any of its parsing algorithms, is that a well-formed PEG makes linear time parsing possible, through a parsing algorithm called packrat parsing [8]. A PEG is well-formed when it does not contain left recursion.

PEG grammars always chose the first alternative matching the remaining input, this could lead to errors that are quite difficult to catch. This, together with the limited support for left recursion, means that one must be very careful when designing a language. More information on PEGs can be found in [9].

5.3 Design choices

The tool is divided into multiple modules, namely:

- `lcsfe-algorithm`,
- `lcsfe-cli`,
- `lcsfe-common`,
- `lcsfe-pg`,
- `lcsfe-mu-ald`.

Module `lcsfe-algorithm` is the core of the project: it contains an implementation of the local algorithm for verifying solutions of systems of fixpoint equations over complete lattices. Module `lcsfe-cli` is a command line interface. Modules `lcsfe-pg` and `lcsfe-mu-ald` both use the local algorithm. As input, they take some specification language and some verification logic. Then they translate this input to a system of fixpoint equations and generate the correct symbolic ∃-moves for their respective operators, after which they call the local algorithm to solve the verification problem, after which the output is passed to `lcsfe-cli` to be printed. Module `lcsfe-common` exposes a common interface that `lcsfe-pg` and `lcsfe-mu-ald` use to provide their results to the command line interface module; it avoids circular dependency.

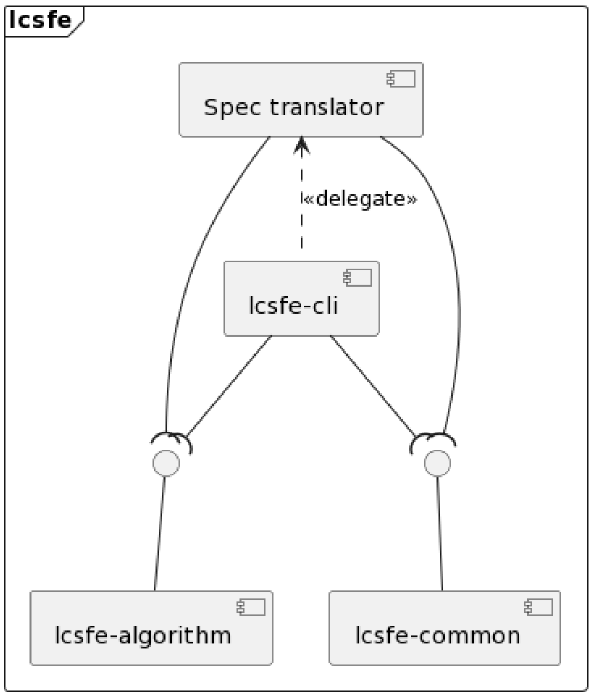

Figure 5.1: A component diagram of LCSFE.

Figure 5.1 represents how the various modules of LCSFE are related. In the diagram, Spec translator represents both `lcsfe-pg` and `lcsfe-mu-ald`. From the diagram we understand that `lcsfe-algorithm` offers an interface, represented by the ball notation, which is accessed by every other module. The `lcsfe-common` crate exposes a trait, represented in the diagram as an interface, via the ball notation. The goal of this trait is to uniform the results computed by Spec translator, so that `lcsfe-cli` can easily access and print them via the same common interface. The Spec translator module is used by `lcsfe-cli`: as input, the former takes a specification file and some verification logic and provides the latter with the results of the computation.

5.3.1 The local algorithm module

Module `lcsfe-algorithm` implements the local algorithm we described in section 4. In the following we show this crate in greater detail.

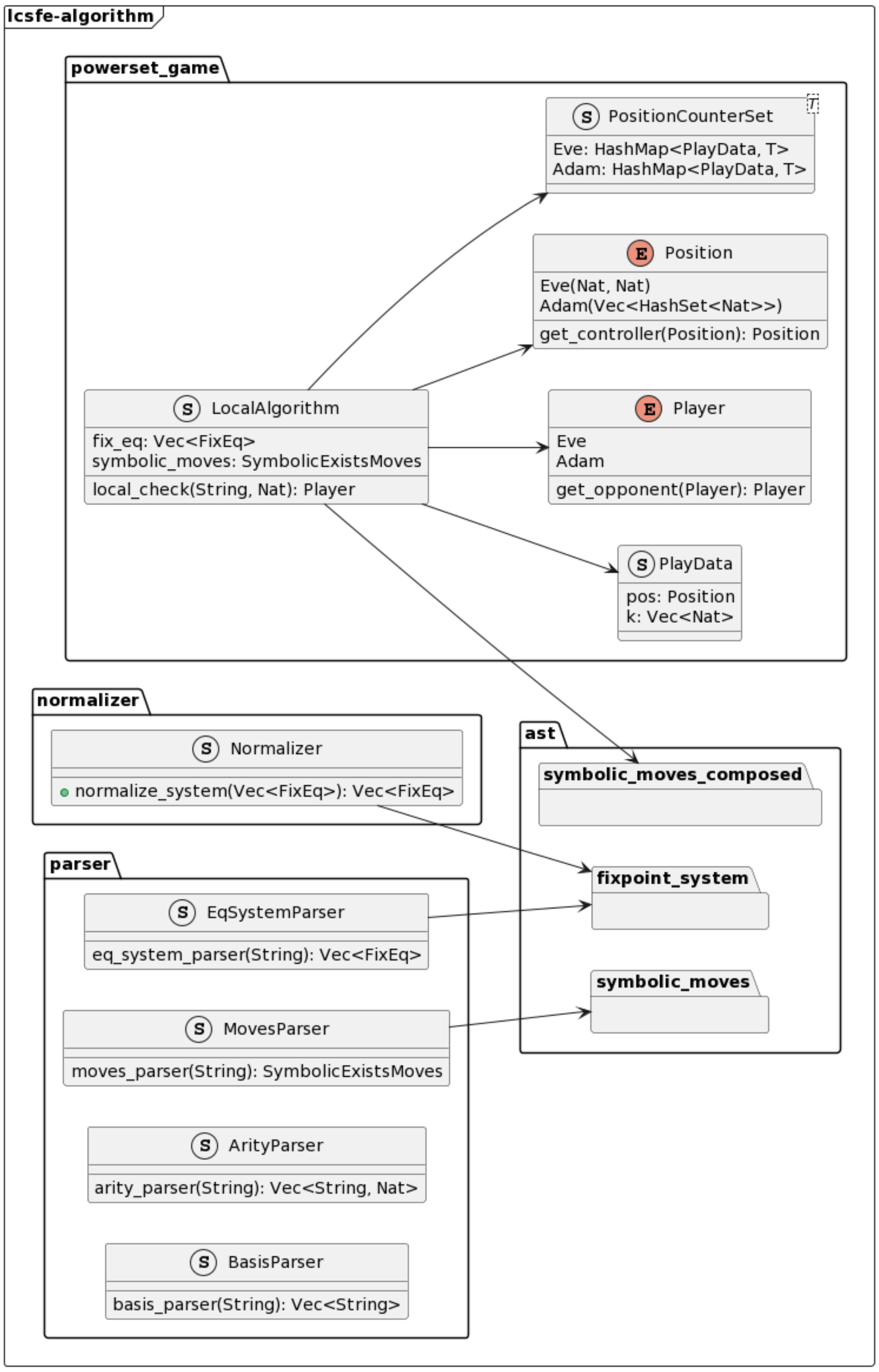
lcsfe-algorithm
powerset_game
PositionCounterSet
T
Eve: HashMap<PlayData, T>
Adam: HashMap<PlayData, T>
Position
Eve(Nat, Nat)
Adam(Vec<HashSet<Nat>>)
get_controller(Position): Position
LocalAlgorithm
fix_eq: Vec<FixEq>
symbolic_moves: SymbolicExistsMoves
local_check(String, Nat): Player
Player
Eve
Adam
get_opponent(Player): Player
PlayData
pos: Position
k: Vec<Nat>
normalizer
Normalizer
normalize_system(Vec<FixEq>): Vec<FixEq>
ast
symbolic_moves_composed
fixpoint_system
symbolic_moves
parser
EqSystemParser
eq_system_parser(String): Vec<FixEq>
MovesParser
moves_parser(String): SymbolicExistsMoves
ArityParser
arity_parser(String): Vec<String, Nat>
BasisParser
basis_parser(String): Vec<String>

Figure 5.2: A class diagram of the module `lcsfe-algorithm`.

Figure 5.2 shows a class diagram of the module. We do not show the structure of the crate in its entirety but only wish to highlight some of its notable features. This diagram roughly represents the interface offered by the crate, meaning that other crates can access the structs, enums and functions represented.

The crate offers its clients a number of functionalities:

- the ability to parse symbolic $\exists$-moves, systems of fixpoint equations, name and arity of the operators in the system and the basis;
- it makes normalizing the system of fixpoint equations possible;
- it offers a representation, or better, abstract syntax trees, of symbolic $\exists$-moves and fixpoint equations;
- finally, it can verify local solutions for systems of fixpoint equations.

5.3.1.1 The local algorithm

In this section we make a thorough description of the module `powerset_game` in Figure 5.2. The local algorithm implementation is wrapped in a structure called `LocalAlgorithm`. A client that wants to solve a verification task must first build this struct by providing a system of fixpoint equations and the corresponding symbolic $\exists$-moves. Then it should call the function `local_check(String, Int): Player`, which takes as input (b, i), a starting position of the existential player, where b is a string and i is a natural number, and returns the winning player from such position. The other functions, such as Explore and Backtrack, are not shown in the diagram: functions shown in Chapter 4 are already quite close to the actual implementation.

Note that there are a number of types in the `powerset_game` module: each can be mapped to a construct described in Chapter 4. Players $\exists$ and $\forall$ are represented by the enum `Player`, player $\exists$ is `Player::Eve`, player $\forall$ is `Player::Adam`. A position is an enum as well. Positions of $\forall$ are elements of a set $Pos_\forall = (2^{B_L})^m$, which are represented in the source code by the enum variant `Position::Adam(Vec<HashSet<Nat>>)`, where where the `Nat` type represents an element of the basis. Positions for the existential player $(b, i) \in B_L \times \underline{m}$ are represented by the enum variant `Position::Eve(Nat, Nat)`, where the first natural number represents a basis element and the second an index. Note that we have a mapping for basis elements into natural numbers. Assumptions and decisions are both represented by structure `PositionCounterSet<T>`, where `T` is a generic data type. In our implementation, for both assumptions and decisions, `T` will be a timestamp. `PlayData` is a struct that contains a position C and a counter $\boldsymbol{k}$, the latter represented by a vector of naturals. We noticed that C and $\boldsymbol{k}$ often appear as a tuple $(C, \boldsymbol{k})$, which is exacly what `PlayData` represents.

A notable feature that we want to highlight is that the function $M^\forall$ is implemented via iterators: iterators in Rust are lazy, which allows us to iterate through elements of an instance of `Position::Adam(Vec<HashSet<Nat>>)`, and extract a move (b, i) whenever needed, rather than computing the whole set of moves from a specific position of $\forall$, and then extracting one.

5.3.1.1.1 INPUT GRAMMARS We now give the grammar, in EBNF form, for systems of fixpoint equations, symbolic $\exists$-moves, basis and the arity of the operators.

$$
\begin{array}{rcl}
\langle \textit{eq\_list} \rangle & ::= & \langle \textit{eq} \rangle\ \langle \textit{eq\_list} \rangle\ \texttt{;} \mid \langle \textit{eq} \rangle\ \texttt{;} \\
\langle \textit{eq} \rangle & ::= & \langle \textit{id} \rangle\ \texttt{=max}\ \langle \textit{or\_exp\_eq} \rangle \mid \langle \textit{id} \rangle\ \texttt{=min}\ \langle \textit{or\_exp\_eq} \rangle \\
\langle \textit{atom} \rangle & ::= & \langle \textit{id} \rangle \mid \texttt{(}\ \langle \textit{or\_exp\_eq} \rangle\ \texttt{)} \mid \langle \textit{custom\_exp\_eq} \rangle \\
\langle \textit{and\_exp\_eq} \rangle & ::= & \langle \textit{atom} \rangle\ (\texttt{and}\ \langle \textit{atom} \rangle)^* \\
\langle \textit{or\_exp\_eq} \rangle & ::= & \langle \textit{and\_exp\_eq} \rangle\ (\texttt{or}\ \langle \textit{and\_exp\_eq} \rangle)^* \\
\langle \textit{custom\_exp\_eq} \rangle & ::= & \langle \textit{op} \rangle\ \texttt{(}\ \langle \textit{or\_exp\_eq} \rangle\ (\texttt{,}\ \langle \textit{or\_exp\_eq} \rangle)^*\ \texttt{)} \\
\langle \textit{id} \rangle & ::= & \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"} \\
\langle \textit{op} \rangle & ::= & \texttt{"}\ (\text{ any ASCII string })\ \texttt{"}
\end{array}
$$

The grammar above represents a system of fixpoint equations. We notice that the syntactic category $\langle \textit{and\_exp\_eq} \rangle$ has a higher precedence than $\langle \textit{or\_exp\_eq} \rangle$; by doing this we enforce the precedence of operator $\wedge$ over $\vee$. String $\langle \textit{op} \rangle$ represents the name of an operator provided by the user. If the goal is to parse μ-calculus formulas, $\langle \textit{op} \rangle$ would accept, for example, strings such as "diamond", or "box". Rule $\langle \textit{custom\_exp\_eq} \rangle$ is a custom operator: all operators in an instance of a system of fixpoint equations will appear as functions with their respective arguments between parentheses, except for operators `and` and `or`, which are conveniently already provided and are infix. A C-style identifier respects the following regex pattern: `[a-zA-Z_][a-zA-Z0-9_]*`.

$$
\begin{aligned}
\langle \mathit{sym\_mov\_list} \rangle &::= \langle \mathit{sym\_mov\_eq} \rangle\ \langle \mathit{sym\_mov\_list} \rangle\ \texttt{;} \mid \langle \mathit{sym\_mov\_eq} \rangle\ \texttt{;} \\
\langle \mathit{sym\_mov\_eq} \rangle &::= \texttt{phi (} \langle \mathit{id} \rangle \texttt{) (} \langle \mathit{num} \rangle \texttt{) = } \langle \mathit{disjunction} \rangle \\
\langle \mathit{conjunction} \rangle &::= \langle \mathit{atom} \rangle\ (\texttt{and}\ \langle \mathit{atom} \rangle)^* \\
\langle \mathit{disjunction} \rangle &::= \langle \mathit{conjunction} \rangle\ (\texttt{or}\ \langle \mathit{conjunction} \rangle)^* \\
\langle \mathit{atom} \rangle &::= \texttt{[} \langle \mathit{id} \rangle \texttt{ , } \langle \mathit{num} \rangle \texttt{]} \mid \texttt{true} \mid \texttt{false} \mid \texttt{(} \langle \mathit{disjunction} \rangle \texttt{)} \\
\langle \mathit{id} \rangle &::= \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"} \\
\langle \mathit{num} \rangle &::= \mathbb{N}
\end{aligned}
$$

The grammar above represents the symbolic ∃-moves for the operators appearing in a system of fixpoint equations. Note that, similarly to what we did for the grammar of systems of fixpoint equations, the conjunction operator has precedence over disjunction operator.

We now give the grammar of a basis: it is simply a list of strings separated by the new-line character `\n`.

$$
\begin{aligned}
\langle \mathit{basis} \rangle &::= \langle \mathit{basis\_elem} \rangle\ \texttt{\textbackslash n}\ \langle \mathit{basis} \rangle \mid \langle \mathit{basis\_elem} \rangle \\
\langle \mathit{basis\_elem} \rangle &::= \texttt{"}\ (\text{ any ASCII string })\ \texttt{"}
\end{aligned}
$$

The grammar specification of a language for specifying operator names and their arity follows.

$$
\begin{aligned}
\langle \mathit{arity} \rangle &::= \langle \mathit{op\_name} \rangle\ \langle \mathit{num} \rangle\ \texttt{\textbackslash n}\ \langle \mathit{arity} \rangle \mid \langle \mathit{op\_name} \rangle\ \langle \mathit{num} \rangle \\
\langle \mathit{op\_name} \rangle &::= \texttt{"}\ (\text{ a C-style identifier })\ \texttt{"} \\
\langle \mathit{num} \rangle &::= \mathbb{N}
\end{aligned}
$$

5.3.1.2 Abstract syntax trees

In this section we describe the data structures we used to represent systems of fixpoint equations, symbolic ∃-moves and symbolic ∃-moves after composition. We decided to use different representations for the symbolic ∃-moves for the operators and for composed symbolic-∃-moves: the former are indexed by the name of a function (the operator), and by a basis element, while the latter by a function in the system of fixpoint equations and a basis element.

5.3.1.2.1 SYSTEMS OF FIXPOINT EQUATIONS In Figure 5.3, we show the structure of a fixpoint equation. Fixpoint equation $x_i =_\eta f_i(\boldsymbol{x})$ is represented by struct `FixEq`. It contains the name of the left-hand side variable, the value of η, i.e., if we want a least or greatest fixpoint, and the function itself, represented by the enum `ExpFixEq`. The right-hand side of a fixpoint formula is represented by the enum `ExpFixEq`, which is either a custom operator or a variable. A custom operator is a recursive type defined by its name and its parameters, that are `ExpFixEq` types as well.

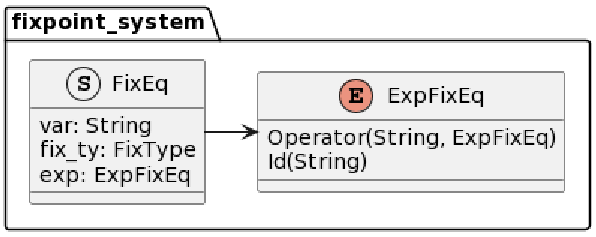

Figure 5.3: Class diagram for the module `fixpoint_system`.

5.3.1.2.2 SYMBOLIC ∃-MOVES Figure 5.4 displays the abstract syntax tree of the symbolic ∃-moves for the operators. Formulas are inside struct `SymbolicExistsMoves`, in vector `formulas`. Moreover, struct `SymbolicExistsMoves` contains mappings of both basis elements and operators' names to natural numbers. It follows that `formulas` can be seen as a matrix with elements of the basis as columns and the operators as rows. Let L be a complete lattice, f a monotone operator $f : L \to L$ and $b \in B_L$; the index of a formula φ_b in vector `formulas` is `fun_map[f] * fun_map.len() + basis_map[b]`, where `fun_map[f]` and `basis_map[b]` are the mappings of f and b to natural numbers and `fun_map.len()` is the number of operators defined. A logic formula has the usual interpretation: it is either an element of the basis, a conjunction or a disjunction (see Definition 20). For the sake of convenience, we decided to add atoms $false$ and $true$, instead of checking whether conjunctions and disjunctions are empty.

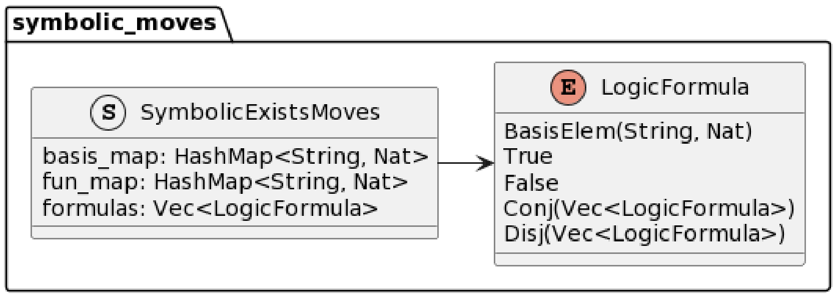

Figure 5.4: Class diagram for the module `symbolic_moves`.

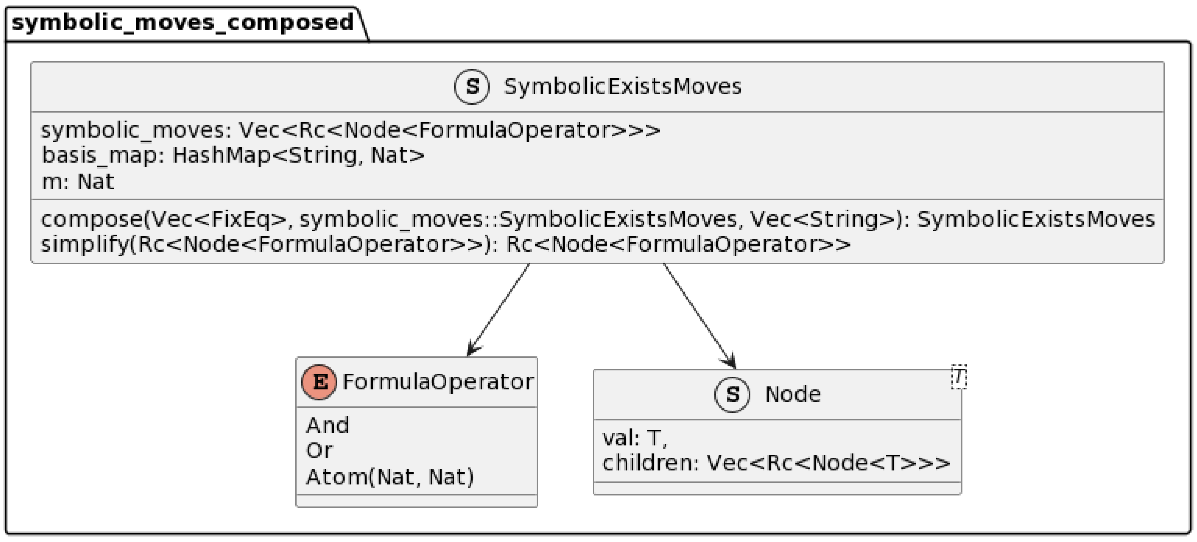
symbolic_moves_composed
S SymbolicExistsMoves
symbolic_moves: Vec<Rc<Node<FormulaOperator>>>
basis_map: HashMap<String, Nat>
m: Nat
compose(Vec<FixEq>, symbolic_moves::SymbolicExistsMoves, Vec<String>): SymbolicExistsMoves
simplify(Rc<Node<FormulaOperator>>): Rc<Node<FormulaOperator>>
E FormulaOperator
And
Or
Atom(Nat, Nat)
S Node
T
val: T,
children: Vec<Rc<Node<T>>>

Figure 5.5: Class diagram for the module `symbolic_moves_composed`.

5.3.1.2.3 Symbolic ∃-moves composed Figure 5.5 shows the internals of module `symbolic_moves_composed`. A composed symbolic ∃-move for a function of a system of fixpoint equations is a tree structure. We show in Figure 5.6 a tree representation of a formula. Symbolic ∃-moves are stored in vector `symbolic_moves` and they are indexed as in the representation of symbolic ∃-moves for operators. A formula is a type `Rc<Node<FormulaOperator>>`, which intuitively is the root of a tree. Type `Rc<T>` is one of Rust's smart-pointer types: it is an immutable pointer to an object in the heap which can be owned multiple times. In simpler terms, it is a pointer that can be copied multiple times, and thus owned multiple times, without copying the object pointed to.

A instance of the struct `Node` contains a value, which in this context is instantiated to `LogicFormula` and a vector of pointers to children nodes. The value of a node is instantiated to enum `LogicFormula`. A `LogicFormula` is either a conjunction or a disjunction operator, or a leaf of the kind $[b, i]$. An atom *true* or *false* is a node without children, i.e., an empty conjunction or an empty disjunction.

We also show the function `compose` which is the constructor of `SymbolicExistsMoves`: as input, it takes a system of fixpoint equations, the operators' symbolic ∃-moves and a basis; it performs the composition and returns an instance of `symbolic_moves_composed::SymbolicExistsMoves`.

Using a tree structure together with type `Rc<T>` removes the need of copying: whenever we need to store a formula φ in the playlist ρ we simply clone the root pointer.

5.3.1.2.4 A DAG data structure for symbolic ∃-moves? Here we propose an optimisation of the algorithm, which should speed up the reduction of a symbolic ∃-move performed by Algorithm 2. This optimisation is not implemented in `LCSFE` due to time constraints. It comes from the observation that in a formula the same subformula can appear more than once. Using the current data structure for composed symbolic ∃-moves, we can build a tree without duplication in atoms of the kind $[b, i]$, just as in Figure 5.6. As of now, the only purpose of this implementation is to improve, timewise, the composition of formulas: each time we need an atom of type $[b, i]$ we retrieve it from a buffer stored in the `SymbolicExistsMoves` struct instead of creating a new atom each time. This is not shown in Figure 5.5. Our intuition is that by having a high degree of sharing in a formula, that is, identical subformulas are represented by a single object in memory, we would improve Algorithm 2 when a symbolic ∃-move contains identical subformulas.

We see this as a future contribution, we provide more details in Section 6.1.

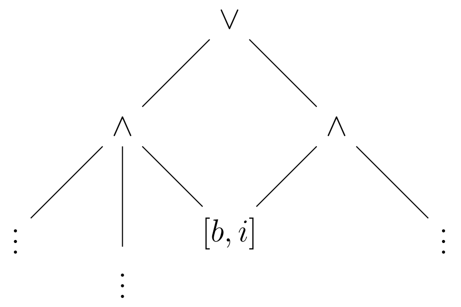

Figure 5.6: Tree representation of a formula, without duplication in the atoms.

5.3.2 The μ-calculus module

We now provide some details about the `lcsfe-mu-ald` module. This module requires two different files as input. The first one is an Aldebaran specification, representing a labelled transitions system, the second one is a μ-calculus formula. The Aldebaran syntax is a simple format for specifying labelled transition systems. It is widely used, most notably by the CADP toolset [11]. Moreover, we require a starting node, from which the model checking of the labelled transition system is carried out.

5.3.2.1 The μ-calculus grammar

We want to parse the following syntax, described with more details in Section 2.4.1.

$$
\begin{aligned}
&A ::= a \mid \mathit{true} \mid \neg a\\
&\varphi ::= \boldsymbol{t} \mid \boldsymbol{f} \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mu x.\varphi \mid \nu x.\varphi \mid \langle A \rangle \varphi \mid [A]\varphi
\end{aligned}
$$

Where $a \in \mathit{Act}$ and $x \in \mathit{PVar}$. For the reasons set out in Section 5.2, we designed a grammar that avoids left recursion as much as possible. The following EBNF grammar describes a μ-calculus formula.

$$
\begin{aligned}
\langle \mathit{atom} \rangle &::= \texttt{tt} \mid \texttt{ff} \mid \texttt{(}\ \langle \mathit{mu\_calc} \rangle\ \texttt{)} \mid \langle \mathit{id} \rangle\\
\langle \mathit{modal\_op} \rangle &::= \texttt{<}\ \langle \mathit{label} \rangle\ \texttt{>}\ \langle \mathit{atom} \rangle \mid\ \texttt{[}\ \langle \mathit{label} \rangle\ \texttt{]}\ \langle \mathit{atom} \rangle\\
\langle \mathit{conjunction} \rangle &::= \langle \mathit{modal\_op} \rangle\ (\texttt{\&\&}\ \langle \mathit{modal\_op} \rangle)\\
\langle \mathit{disjunction} \rangle &::= \langle \mathit{conjunction} \rangle\ (\texttt{||}\ \langle \mathit{conjunction} \rangle)\\
\langle \mathit{fix\_op} \rangle &::= \texttt{mu}\ \langle \mathit{id} \rangle\ \texttt{.}\ \langle \mathit{disjunction} \rangle \mid\ \texttt{nu}\ \langle \mathit{id} \rangle\ \texttt{.}\ \langle \mathit{disjunction} \rangle\\
\langle \mathit{mu\_calc} \rangle &::= \langle \mathit{fix\_op} \rangle \mid \langle \mathit{disjunction} \rangle\\
\langle \mathit{label} \rangle &::= \texttt{true} \mid \langle \mathit{id} \rangle \mid \texttt{!}\ \langle \mathit{id} \rangle\\
\langle \mathit{id} \rangle &::= \texttt{"}\ \text{(a C-style identifier)}\ \texttt{"}
\end{aligned}
$$

Moreover, we designed this grammar to respect some standard conventions: modal operators $[a]$ and $\langle a \rangle$ bind stronger than $\vee, \wedge$, and the fixpoint operators capture everything after the `.` character. The consequence is that a formula $\mu x(([a]x) \vee \nu y(\langle a \rangle y \wedge \boldsymbol{f}))$ can be written as $\mu x.[a]x \vee \nu y.\langle a \rangle y \wedge \boldsymbol{f}$, minimizing the use of parenthesis. Whenever we wish to add anything different from the syntactic categories $\boldsymbol{t}$, $\boldsymbol{f}$ or $x \in \mathit{PVar}$ to a modal operator, parentheses must be used, due to the inherent limitations of the type of parser we used. This is expressed by the rule $\langle \mathit{atom} \rangle$.

5.3.2.2 Symbolic $\exists$-moves for the μ-calculus

From a labelled transition system and a μ-calculus formula, we can compute the symbolic $\exists$-moves for the μ-calculus operators. Here we provide a definition for each operator, with respect to the semantics described in Section 2.4. For each $b \in B_L$, with L the powerset lattice of the set of states $\mathbb{S}$, represented by the nodes in the transition system:

- $\varphi_b^{\cup} = [b, 1] \vee [b, 2]$, for $\cup : L \times L \to L$;
- $\varphi_b^{\cap} = [b, 1] \wedge [b, 2]$, for $\cap : L \times L \to L$;
- $\varphi_b^{[a]} = \bigwedge\{\, [x, 1] \mid b \xrightarrow{a} x \,\}$, for each $a \in \mathit{Act}$, $[a] : L \to L$;
- $\varphi_b^{\langle a\rangle} = \bigvee\{\, [x, 1] \mid b \xrightarrow{a} x \,\}$, for each $a \in \mathit{Act}$, $[a] : L \to L$;
- $\varphi_b^{\boldsymbol{t}} = \mathit{true}$;
- $\varphi_b^{\boldsymbol{f}} = \mathit{false}$.

In a formula where the operator $[A]$ or $\langle A\rangle$ appears, with $A = \mathit{true}$, a symbolic $\exists$-move for each $a \in \mathit{Act}$ must be generated. If otherwise $A = \neg a$ we generate all symbolic $\exists$-moves for the modal operators, besides for operators $[a], \langle a\rangle$.

5.3.3 The parity game solver

We now provide some details about `lcsfe-pg`. As input, this module receives a file which represents a parity game, which is then translated into a system of boolean fixpoint equations. A system of boolean fixpoint equations is a system of fixpoint equations over the complete lattice $\mathit{false}, \mathit{true}$, where $\mathit{false} \sqsubseteq \mathit{true}$ and only operators $\wedge$ and $\vee$ appear. We show a translation of a parity game into a system of boolean fixpoint equations in Example 6. The user is also required to provide a starting node, from which the local algorithm then search for a winning strategy for one of the players.

5.3.3.1 The PGSolver format

The parity game solver uses PGSolver's grammar, which we show here with a brief description.

$$
\begin{aligned}
\langle \textit{parity\_game} \rangle &::= [\texttt{parity}\ \langle \textit{identifier} \rangle\ \texttt{;}]\ \langle \textit{node\_spec} \rangle^{+} \\
\langle \textit{node\_spec} \rangle &::= \langle \textit{identifier} \rangle\ \langle \textit{priority} \rangle\ \langle \textit{owner} \rangle\ \langle \textit{successors} \rangle\ [\langle \textit{name} \rangle]\ \texttt{;} \\
\langle \textit{identifier} \rangle &::= \mathbb{N} \\
\langle \textit{priority} \rangle &::= \mathbb{N} \\
\langle \textit{owner} \rangle &::= \texttt{0} \mid \texttt{1} \\
\langle \textit{successors} \rangle &::= \langle \textit{identifier} \rangle\ (\texttt{,}\ \langle \textit{identifier} \rangle)^{*} \\
\langle \textit{name} \rangle &::= \texttt{"}\ (\text{ any ASCII string not containing `}\texttt{"}\text{'})\ \texttt{"}
\end{aligned}
$$

The ⟨*identifier*⟩ in ⟨*parity_game*⟩ should be the maximum priority of the parity game described in the file. In PGSolver it is used for a more efficient parsing. It is ignored in the parser we have developed. Each line in the file describes a node, uniquely identified by ⟨*identifier*⟩, and optionally a string. Each node is then coupled with its priority, owner, and a list of successors.

5.3.3.2 Symbolic ∃-moves for systems of boolean fixpoint equations

From the parity game specification we generate a system of boolean fixpoint equations. The next step is to extract, or rather, compose, the corresponding symbolic ∃-moves. Note that in a system of boolean fixpoint equations only operators $\land$ and $\lor$ appear. For $b \in B_L$, where L is the boolean lattice $\{ \mathit{false}, \mathit{true} \}$, with $\mathit{false} \sqsubseteq \mathit{true}$, and $B_L = \{ \mathit{true} \}$, the symbolic ∃-moves for both operators are the following:

- $\varphi_b^{\land} = [b, 1] \land [b, 2]$, for $\land : L \times L \to L$;
- $\varphi_b^{\lor} = [b, 1] \lor [b, 2]$, for $\lor : L \times L \to L$.

5.4 Examples and performance considerations

In this section we show three examples of executions of `LCSFE`. We solve the same three examples with Oink and mCRL2. We use them to discuss the performance of our tool. All tests are performed on the same machine: a laptop powered by an AMD Ryzen 5 5500 processor and 8 gigabytes of RAM, with the Linux kernel 6.5.11.

We use the following parity game, which is the same in Figure 2.1.

```
parity 4;
0 6 1 4,2 "Africa";
```

```
4 7 1 0 "Antarctica";
1 5 1 2,3 "America";
3 6 0 4,2 "Australia";
2 8 0 3,1,0,4 "Asia";
```

We want to know whether player ∃ wins from node `Antarctica`: to do that we run the following command.

```
> cargo run -r -- pg tests/parity_games/test_03.gm Antarctica
```

File `test_03.gm` contains the game specification. The command `cargo run -r` compiles and runs `LCSFE` in release mode, which creates an optimised executable. The compilation might take a few seconds. Then, aside from the compilation messages, the output is the following.

```
Preprocessing took: 0.000022963 sec.
Solving the verification task took: 0.000010881 sec.
Result: Player 1 wins from vertex Antarctica
```

There is a preprocessing phase before running the local algorithm. The preprocessing phase encompasses extracting the fixpoint system from the specification and generating and composing the symbolic ∃-moves. For parity games, the preprocessing phase consists in extracting the system of boolean fixpoint equations and composing the moves, there are only two operators in a system of boolean fixpoint equations, $\wedge$, $\vee$ and we provide symbolic ∃-moves for both by default. In the case of `lcsfe-mu-ald`, the preprocessing comprises extracting the system of fixpoint equations from the μ-calculus formula, generating the symbolic moves for each operator appearing in the formula, instantiated to the labelled transition system provided as input and composing them to symbolic ∃-moves.

We solve the same parity game on the same machine with Oink, via the following command.

```
> oink tests/parity_games/test_03.gm -p
```

The output of the command is shown below.

```
[    0.00] parity game with 5 nodes and 11 edges.
[    0.00] parity game reindexed
[    0.00] parity game renumbered (4 priorities)
[    0.00] no self-loops removed.
[    0.00] 2 trivial cycles removed.
[    0.00] preprocessing took 0.000017 sec.
[    0.00] solved by preprocessor.
[    0.00] total solving time: 0.000033 sec.
[    0.00] current memory usage: 4.62 MB
[    0.00] peak memory usage: 4.75 MB
[     0.00] won by even: America Asia Australia
[     0.00] won by odd: Africa Antarctica
```

In this very simple example, `LCSFE` performance is on par with Oink's, even though Oink finds the global solution, instead of just a winner from a node.

The next examples come from the mCRL2 tutorial, which can be found at the following link: https://www.mcrl2.org/web/user_manual/tutorial. We want to solve two problems. The first problem we solve is "The Rope Bridge": we want to know whether four adventurers can cross a bridge in 17 minutes. We have the following constraints: no more than two persons can cross the bridge at once and a flashlight needs to be carried by one of them every crossing. The adventurers have only one flashlight at their disposal. They are not all equally skilled: crossing the bridge takes them 1, 2, 5 and 10 minutes, respectively. A pair of adventurers cross the bridge in an amount of time equal to that of the slowest of the two adventurers. We define the following μ-calculus formula: $\mu X.\boldsymbol{t} \vee \langle true\rangle(\langle report(17)\rangle X)$.

To use a mCRL2 specification in `LCSFE`, we convert it to a labelled transition system in Aldebaran format, using the tool `ltsconvert`, from mCRL2's toolset. Then we run our tool. The resulting transition system has 102 states, and 177 transitions.

```
> cargo run -r -- mu-ald tests/example_mucalc/bridge-referee.aut \
  tests/example_mucalc/receive-17 0
```

We pass as input to the command line interface the Aldebaran specification file, `bridge-referee.aut`, and the file containing the μ-calculus formula. We want to perform local model checking from state 0 of the labelled transition system. Follows the output of the command.

```
Preprocessing took: 0.000235959 sec.
Solving the verification task took: 0.000004718 sec.
Result: The property is satisfied from state 0
```

To do the same with mCRL2, we run the following commands. We first need to translate the mCRL2 specification the to `.lps`, which is a file format used internally by mCRL2.

```
> mcrl22lps bridge-referee.mcrl2 bridge.lps
```

The command below takes a μ-calculus formula, in `formula_A-final.mcf`, and the file we have just generated, `bridge.lps`, and builds a type of system of boolean fixpoint equations, called parameterised boolean equation system. This process took 0.017395 seconds to finish.

```
> lps2pbes --formula=formula_A-final.mcf bridge.lps bridge.pbes --timings
```

Finally, we solve the model checking task via the following command.

```
> pbes2bool -rjittyc bridge.pbes --timings
```

The output is the following.

```
true
- tool: pbes2bool
  timing:
    instantiation: 0.009156
    solving: 0.000032
    total: 0.038423
```

In this small instance our tool actually performs better than mCRL2. That is because during the composition of symbolic $\exists$-moves we are able to simplify every formula to *true*: in the μ-calculus formula there is a disjunction and one of the disjunct is $\boldsymbol{t}$.

We now describe the next specification: "Gossips". There are five agents, each have an information that must be shared to all of the others. Agents share information by making phone calls, and each time they share every secret they have come to know. We want to check whether this system is deadlock free. To do so we use the following μ-calculus formula: $\varphi = \nu X.\langle true\rangle \boldsymbol{t} \wedge [true]X$. The result of the conversion of the mCRL2 specification to Aldebaran format is a labelled transition system with 9152 states and 183041 transitions.

```
> cargo run -r -- mu-ald tests/example_mucalc/gossips.aut \
  tests/example_mucalc/deadlock-liveness 0
```

As input, we pass to the command line interface the Aldebaran specification file, `gossips.aut` and the file containing the μ-calculus formula. We want to perform local model checking from state 0 of the labelled transition system.

```
Preprocessing took: 0.16171992 sec.
Solving the verification task took: 5.695183 sec.
Result: The property is satisfied from state 0
```

We run the same commands as before, until we reach the following output.

```
true
    - tool: pbes2bool
      timing:
        instantiation: 1.397916
        solving: 0.002444
        total: 1.424587
```

We see that now mCRL2 is much faster. Moreover, during the execution of `LCSFE` we experienced a stack overflow: due to the recursive nature of the local algorithm we employ the recursive calls filled a stack of 8 megabytes of size. In order to solve this verification task we incremented the size of the stack.

6
Conclusions

In this thesis we explored a game-theoretic view of systems of fixpoint equations over complete lattices, which arise in several verification tasks. The idea is to exploit a game theoretical characterisation of the solution of a system of fixpoint equations in terms of a parity game called powerset game. In view of an implementation, in order to improve the efficiency of the verification process one can introduce a notion of selection, which allows us to restrict the moves of one of the two players, namely, the existential player, without losing the completeness and correctness of the game. Moreover, a logic can be used to represent the moves of the existential player symbolically. The semantics of this logic is designed in a way that there is a connection between formulas of this logic and selections. Concretely, from a formula, we are able to generate moves for the existential player. This thesis contributes to the theoretical framework described in [2] and [17]. We proved that symbolic $\exists$-moves, in Definition 22, can be composed: we can build a symbolic $\exists$-move for a function that combines many monotone operators, provided that we have a symbolic $\exists$-move for each one of them. We also showed that by using the logic we described in Definition 20, we can always represent sets that are upwards-closed with respect to $\sqsubseteq_{\mathrm{H}}^{\wedge}$.

Inspired by the theory in [2] and [17], we proposed a concrete local algorithm for verifying solutions of systems of fixpoint equations over complete lattices. Even though some description of the basic procedures was already given in [2], [17] here we presented the algorithm in a much more detailed way. We first defined the various components of the algorithm using pseudocode. This was the first step towards an actual implementation. We built a tool, `LCSFE`, which shows two applications of our theory, on parity games and on the μ-calculus over labelled transition system. We decided not to limit our tool to only the verification of local solutions for systems of boolean fixpoint equations and systems stemming from μ-calculus formulas. In fact, the user can input any monotone operator over a complete lattice, provided that it is possible to define a symbolic ∃-move for it. By doing this we are able to encode in our theoretical framework many types of verification logics.

6.1 Future contributions

Unfortunately, due to time constraints we are not able to build a benchmark suite for properly exploring the performance of our tool. We were able to observe that our recursive algorithm, described in Chapter 4, can cause stack overflow on examples of a certain size. Thus, as a future improvement, it could be useful to convert the local algorithm, which is currently recursive, to an iterative form.

Improvements to `LCSFE` could also come in the form of a smarter data structure for representing symbolic ∃-moves. In Section 5.3.1.2.4 we discussed a possible optimization for simplifying a formula during the execution of the local algorithm. This optimisation would require using a DAG (Directed Acyclic Graph) structure: ideally, we would like to represent a formula with some degree of sharing between nodes. The goal is to modify at once identical subformulas in a symbolic ∃-move. Whenever we modify a formula in Algorithm 2, identical subformulas, which in a DAG would be represented by the same object in memory, are modified at once. In the current implementation, whenever we need a formula during the execution of the local algorithm, we retrieve the root node. We cannot directly modify any of the nodes of a formula, since otherwise we would modify the original formula, which we might need more than once. Thus, we build an entirely new tree each time we use the Reduce function, and we reuse the nodes that do not need modifications. This, arguably, requires the least number of copies. Intuitively, we are modifying pointers each time we modify a formula. Instead, this optimisation requires us to modify the object referenced by the pointer directly, which might be pointed to by many other nodes. The consequence is that whenever we prune an atom $[b, i]$ from the tree of the formula, i.e., when $[b, i]$ becomes either $true$ or $false$ after an execution of Algorithm 3, all atoms $[b, i]$ are changed at once. This optimization could then be extended to entire subformulas during the invocation of Simplify by Algorithm 2. That is, if a formula contains identical subformulas, once we find that a subformula can be reduced to an atom $true$ or $false$, they are all modified at once. In practice, we think that this kind of optimisation would require significant modifications to the data structure used for composed symbolic ∃-moves, and that it would require using a persistent data structure [7] along with the DAG. By using a persistent data structure, we could effectively modify formulas without causing side-effects on the original symbolic ∃-moves. We did not find instances of formulas with identical subformulas in our pool of examples from parity games and μ-calculus properties. Still, we think that this could very well happen in other types of verification logics.

Additionally, even though we can rely on the original proof to make an informal argument explaining how it can be adapted to the new version of the algorithm (see Section 4), a completely formal proof of correctness would be desirable.

As mentioned before, systems of fixpoint equations are ubiquitous in the setting of formal system verification, ranging from behavioural equivalence (like bisimilarity) to various kinds of qualitative and quantitative logics in the style of the μ-calculus, to static analysis. Hence, it could be interesting to explore the use of this theoretical framework in a number of other contexts. Including a new specification formalism into the tool will require a module that given a system and a property of interest constructs the corresponding system of equations and the associated symbolic $\exists$-moves, as we already did for the μ-calculus and for parity games. For instance, one might be interested in quantitative models and properties, capturing features like time, probabilities or costs. The Łukasiewicz μ-calculus, considered in [18] as a precursor to model-checking PCTL or probabilistic μ-calculi is a type of probabilistic logic. The Łukasiewicz logic can be encoded to a system of fixpoint equations over the set of reals $[0, 1]$. Since quantitative models of this kind typically have an unlimited number of states, we need to define suitable abstractions: an example is a form of discretisation where we consider a fixed number of elements from $[0, 1]$. This topic is developed further in [2].

Another area from which improvements could come are up-to functions. Up-to functions can be applied to systems of fixpoint equations, while preserving the solution of the original system. Moreover, we can enhance our algorithm by using up-to functions, while keeping soundness and correctness. This idea has already been studied extensively in [2]. Further study is needed in order to integrate up-to functions in our local algorithm, more precisely, the question of whether it is possible, and if so how, to represent such functions through symbolic moves.

References

[1] Samson Abramsky and Achim Jung. “Domain Theory”. In: *Handbook of Logic in Computer Science (Vol. 3): Semantic Structures*. USA: Oxford University Press, Inc., 1995, pp. 1–168. ISBN: 019853762X.

[2] Paolo Baldan, Barbara König and Tommaso Padoan. “Abstraction, Up-to Techniques and Games for Systems of Fixpoint Equations”. In: *CoRR* abs/2003.08877 (2020). arXiv: 2003.08877. URL: https://arxiv.org/abs/2003.08877.

[3] Paolo Baldan et al. “Fixpoint games on continuous lattices”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 26:1–26:29. DOI: 10.1145/3290339. URL: https://doi.org/10.1145/3290339.

[4] Patrick Cousot and Radhia Cousot. “Constructive versions of Tarski’s fixed point theorems”. en. In: *Pacific Journal of Mathematics* 82.1 (May 1979), pp. 43–57. ISSN: 0030-8730, 0030-8730. DOI: 10.2140/pjm.1979.82.43. URL: http://msp.org/pjm/1979/82-1/p04.xhtml (visited on 19/10/2023).

[5] Giorgio Delzanno and Jean-François Raskin. “Symbolic Representation of Upward-Closed Sets”. In: *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*. Ed. by Susanne Graf and Michael I. Schwartzbach. Vol. 1785. Lecture Notes in Computer Science. Springer, 2000, pp. 426–440. DOI: 10.1007/3-540-46419-0_29. URL: https://doi.org/10.1007/3-540-46419-0%5C_29.

[6] Tom van Dijk. “Oink: An Implementation and Evaluation of Modern Parity Game Solvers”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 291–308. ISBN: 978-3-319-89960-2.

[7] James R. Driscoll et al. “Making data structures persistent”. In: *Journal of Computer and System Sciences* 38.1 (1989), pp. 86–124. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(89)90034-2. URL: https://www.sciencedirect.com/science/article/pii/0022000089900342.

[8] Bryan Ford. “Packrat Parsing: Simple, Powerful, Lazy, Linear Time”. In: *CoRR* abs/cs/0603077 (2006). arXiv: cs/0603077. URL: http://arxiv.org/abs/cs/0603077.

[9] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation”. en. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Venice Italy: ACM, Jan. 2004, pp. 111–122.

ISBN: 9781581137293. DOI: 10.1145/964001.964011. URL: https://dl.acm.org/doi/10.1145/964001.964011 (visited on 22/11/2023).

[10] Oliver Friedmann and Friedmann Lange. *The PGSolver Collection of Parity Game Solvers*. Version release 4.1. 2022. URL: https://github.com/tcsprojects/pgsolver.

[11] Hubert Garavel et al. “CADP 2011: a toolbox for the construction and analysis of distributed processes”. en. In: *International Journal on Software Tools for Technology Transfer* 15.2 (Apr. 2013), pp. 89–107. ISSN: 1433-2787. DOI: 10.1007/s10009-012-0244-z. URL: https://doi.org/10.1007/s10009-012-0244-z (visited on 25/11/2023).

[12] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. en. The MIT Press, 2014. ISBN: 9780262321037. DOI: 10.7551/mitpress/9946.001.0001. URL: https://direct.mit.edu/books/book/4007/modeling-and-analysis-of-communicating-systems (visited on 10/11/2023).

[13] Marcin Jurdziński. “Small Progress Measures for Solving Parity Games”. In: *STACS 2000*. Ed. by Horst Reichel and Sophie Tison. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 290–301. ISBN: 978-3-540-46541-6.

[14] Dexter Kozen. “Results on the Propositional mu-Calculus”. In: *Theor. Comput. Sci.* 27 (1983), pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6. URL: https://doi.org/10.1016/0304-3975(82)90125-6.

[15] Giovanni Mazzocchin. *PMModelChecker*. 2019. URL: https://github.com/Cyofanni/PMModelChecker.

[16] Giovanni Mazzocchin. “Solving fixpoint equations using Progress Measures”. MA thesis. University of Padua, Department of Mathematics, Apr. 2019.

[17] Denis Mazzucato. “Solving systems of fixpoint equations: an algorithmic perspective”. MA thesis. University of Padua, Department of Mathematics, Sept. 2020.

[18] Matteo Mio et al. “Łukasiewicz μ-Calculus”. In: *Fundam. Inf.* 150.3–4 (Jan. 2017), pp. 317–346. ISSN: 0169-2968. DOI: 10.3233/FI-2017-1472. URL: https://doi.org/10.3233/FI-2017-1472.

[19] Flemming Nielson, Hanne Riis Nielson and Chris Hankin. *Principles of program analysis*. Springer, 1999. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6. URL: https://doi.org/10.1007/978-3-662-03811-6.

[20] Perdita Stevens and Colin Stirling. “Practical Model-Checking Using Games”. In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS ’98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 85–101. DOI: 10.1007/BFb0054166. URL: https://doi.org/10.1007/BFb0054166.

[21] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications”. en. In: *Pacific Journal of Mathematics* 5.2 (June 1955), pp. 285–309. ISSN: 0030-8730, 0030-

8730. DOI: 10.2140/pjm.1955.5.285. URL: http://msp.org/pjm/1955/5-2/p11.xhtml (visited on 19/10/2023).