**Università degli Studi di Padova**

Dipartimento di Matematica
*Corso di Laurea in Informatica*

# Solving fixpoint equations using Progress Measures

*Supervisor*
**Prof. Paolo Baldan**

*Author*
**Giovanni Mazzocchin**

DIPARTIMENTO MATEMATICA

April 2019

# Credits

*Ringraziamenti*
Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Paolo Baldan, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro. Desidero ringraziare con affetto la mia famiglia per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

*Padova, April 2019* Giovanni Mazzocchin

# Abstract

A number of verification and analysis techniques can be expressed in terms of (systems of) fixpoint equations over suitably defined lattices of information. This is the case for many data flow analyses in the setting of static program analysis, behavioural equivalences in reactive systems and verification techniques for temporal logics (especially for the *$\mu$-calculus*). Some recent researches showed that a game-theoretic characterisation of the solution of equational systems can be devised. Such characterisation is given in terms of a *parity game* (a two-player, zero-sum game), which is referred to as *fixpoint game*. Relying on this theoretical basis, this thesis aims at developing a generic game-theoretic algorithm for finding the solution of a given system of equations. We think that such algorithm can be profitably used for all the aforementioned verification and model checking tasks (as we will see, in the setting of model checking, the winner of the fixpoint game determines whether a logical formula is satisfied by a given model).
The notion of *Progress Measure*, a tool first devised by Jurdziński for solving parity games efficiently, will be paramount in establishing which player has a winning strategy in the play.
The main contribution of this thesis will be a generic algorithm and a corresponding tool that, relying on a description of a set of basic operators over some finite lattice, is automatically instantiated to a solver for systems of fixpoint equations over the given lattice.

# Contents

# List of Figures

CHAPTER 1

# Introduction

Systems of fixpoint equations frequently arise in many program analysis and verification tasks. Some fundamental analyses, such as *Available Expressions* and *Reaching Definitions* in static analysis boil down to systems of mutually recursive fixpoint equations generated from the program's flow graph. In the field of reactive systems, *Bisimilarity* is obtained as the approximation of a greatest fixpoint. It should be noted that such analyses do not engender any alternation between least and greatest fixpoints: as a matter of fact, all their equations require either the smallest or the largest solution.

However, this alternation exists in some verification tasks: in fact, specification modal logics such as the $\mu$-*calculus* get most of their power from a profitable mixture between fixpoints used to encode *liveness* and *safety* properties within a concise formula, which in turn is to be translated into a system of equations. Consequently, a logical formula can be *model checked* on a given transition system by veryifing whether a state of such system (often the initial one) is contained in the solution of the system of equations stemming from the formula. *Model checking* refers to situations in which we are given a model of a system and we need to check automatically whether this model meets a specification, which is often expressed in terms of a formula in some logical language.

Usually, the model checking problem is dealt with in terms of a *parity game* (also called *fixpoint game* in the case we studied), whose purport is to provide a correct and complete characterisation of the solution of a given system of equations.
As for parity games, some ideas on the subject are provided so as to let the reader grasp the intuition behind their connection to model checking. A parity game is played by two players, player $\exists$ and player $\forall$ (many alternative denominations are found in the literature), on a graph with priorities assigned to vertices. The play could either terminate or last forever: such cases require different conditions to *solve* the game. The problem of *solving* (or *deciding*) a parity game consists in deciding whether player $\exists$ has a winning strategy from a vertex. Besides, we remark that each of the players either wins or loses, since parity games belong to the larger family of *zero-sum* games. Over the years, algorithms purporting to solve/decide parity games have been devised and ameliorated in terms of complexity. With respect to the first such algorithm, designed by Jurdziński in his seminal paper ([13]), its complexity is polynomial in the number of states and exponential in half

of the formula's alternation depth.
One of the key ingredients of Jurdziński's contribution to the theory of parity games is the notion of *Game Progress Measure*, a mathematical tool that can witness winning strategies in parity games. An interesting aspect of Progress Measures is their characterisation as pre-fixpoints of certain monotone operators on a complete lattice: this result permits to verify the existence of the Progress Measure by means of a straightforward computation. Even though Progress Measures were first devised in a study about parity games, their usefulness goes much beyond this field. Indeed, it has been observed ([12]) that the use of Progress Measures can be generalised to systems of fixpoint equations over general lattices. Especially, the view presented in [12] is the following: Progress Measures are to nested/alternating fixpoints what *invariants* are to safety/greatest fixpoints, and what *ranking functions* are to liveness/least fixpoints.
However, there is not a full match between the definition of Progress Measure given by Jurdziński and the one provided in [12], since the former describes an algorithm for actually computing such Progress Measures, whereas the latter only provides a constructive characterisation of Progress Measures in the case of powerset lattices (which is a big restriction for common program analysis tasks).
In order to overcome the aforementioned limitations, [2] extended the game-theoretic approach to fixpoint equations to a wider category of lattices, the so-called *continuous lattices*, which were first studied under this name in [21]. Examples of continuous lattices occur in many areas of algebra, analysis and topology. They are usually defined in terms of the so-called *way-below* relation, which is definable in any complete lattice. D.S. Scott introduced the notion of continuous lattice as a basis for an abstract theory of computation.

The theoretical framework devised in [2] provides an extension to the well-known characterisation of the fixpoint of a single, monotone function by means of a game between an existential and a universal player. In particular, this game-theoretic approach is lifted from single functions/equations to systems of fixpoint equations. A detailed explanation of the basic game-theoretic approach to model check a single formula can be found in [25], where the author describes an *evaluation game* $\mathcal{E}(\xi, \mathbb{S})$ (an instance of a *board game*, in which the players move a token along the edge relation of some graph) associated with a fixed formula $\xi$ and a fixed LTS $\mathbb{S}$. It should be noted that an extension of the game to equational systems forces us to face issues caused by the mix of least and greatest fixpoint equations. One of such issues is the need for a novel, non-trivial winning condition.
The core of the work by [2] is undoubtedly the introduction of the notion of Progress Measure for fixpoint games over continuous lattices. Though an entire chapter of our thesis has been devoted to this concept, some intuition can already be given here. Consider a position $(b, i)$ of the existential player in the fixpoint game, that is, an element $b$ of the lattice's basis and an equation index $i$ (recall that the game is played on systems): the Progress Measure is a function returning a vector of *ordinals* specifying how many iterations are needed for each equation to cover $b$ in the $i$-th component of the solution. [2] provides also a characterisation of Progress Measures as least fixpoints, a fundamental feature that gives us a constructive way to compute them and, as a consequence, solve the game. After observing that the current definition of Progress Measure leads to inefficiency in their computation, [2]

presents the notion of *selection*, which essentially puts some contraints on the moves of the existential player in the fixpoint game, thus reducing the number of moves considered for computing Progress Measures. Selections are thus expressed using an ad-hoc logical language able to represent upward-closed sets (we will see that the set of moves of each player is an upward-closed set) in a very concise manner. Such logical language is finally used to give an alternative, more efficient formulation for Progress Measures. Finally, in order to speed up the computation, any suitable optimisation of fixpoint calculation from the literature should be considered and put into practice.

Now, let us go back to the objective of this work. We move from the observation that the theory outlined above, when the lattice of interest is finite, suggests an effective procedure for characterising the solution of a system of fixpoint equations. The aim is to translate such intuition into practice by devising a generic algorithm for the solution of systems of equations over a finite, complete lattice, with an eye to efficiency. The keystone consists in finding efficient fixpoint algorithms for solving the Progress Measure equations described in [2]. Afterwards, some further optimisations for the solution of fixpoint equational systems are experimented. One of such improvements is sketched by Jurdziński himself: in fact, he talks explicitly about better strategies for computing Progress Measures in his seminal paper, which could include some clever ways to implement the Worklist Algorithm, a fundamental general procedure in program analysis which can be instantiated in several ways.
As regards other viable optimisations, we mention the one introduced in [6], which develops an algorithm for model checking that handles the full modal $\mu$-calculus including alternating fixpoints, characterising formulae in terms of equational systems (which is exactly the setting we are working on). This optimisation tries to achieve a better time complexity by developing the notion of *closed subsystem* of an equational system; intuitively, such closed subsystems derive from the *strongly-connected components* of the dependency graph associated to the system of equations. Within a strongly-connected component each pair of variables is mutually dependent, while there can exist at most a hierarchical dependency between two variables in distinct strongly-connected components.
It is known that any system of fixpoint equations can be translated into an equivalent system which includes only *normalised* equations. A normalised equation is an equation whose right-hand side does not contain a composition of operators. A natural question therefore arises: is the solution of a normalised system faster? Observe that the different efficiency in the solution of a normalised system could derive from the fact that the system's equations are solved on different product lattices.

Trying to respond to each of the above-mentioned efficiency-related issues requires a real tool able to process a system and output the solution of its associated system of Progress Measure equations. Even though our final result does not compete with existing solvers in terms of efficiency, the tool we designed provides a useful and extensible proof of concept for the theoretical framework devised in [2].

The rest of the thesis is organised as follows:

* Chapter 2 contains a brief overview of lattice theory, systems of fixpoint equations, $\mu$-calculus and parity games;
* Chapter 3 is an in-depth report about the theory of fixpoint games and Progress Measures. Some efficiency issues related to Progress Measures are dealt with thoroughly;
* Chapter 4 includes the most important algorithms implemented in the tool. In particular, fixpoint algorithms are dealt with in detail;
* Chapter 5 contains a detailed description of the final product, from its design to the implementation of each of its components;
* Chapter 6 concludes the thesis and outlines future improvements.

CHAPTER 2

# Theoretical Background

In this chapter we provide a synopsis of the main theoretical topics our research is based on. This thesis relies heavily on *Lattice Theory* as well as on some aspects of $\mu$-calculus, and lastly on algorithmic research issues on parity games.
We therefore devoted a distinct section to each topic, in order to encompass all the theory needed to understand the remainder of the work.

## 2.1 Lattices

Lattices constitute undoubtedly the most important mathematical structure treated in this thesis. Thus, a list of essential definitions and theorems about this subject was thought to be helpful.

**Definition 2.1.1.** *(**Complete lattice, Upward-closure, Basis, Irreducibles**).*
*A* lattice *is a partially ordered set* $(L, \sqsubseteq)$ *such that each pair,* $l_1$, $l_2 \in L$ *admits a join* $l_1 \sqcup l_2$ *and a meet* $l_1 \sqcap l_2$.
*A* complete lattice *is a partially ordered set* $(L, \sqsubseteq)$ *such that each subset* $X \subseteq L$ *admits a join* $\bigsqcup X$ *and a meet* $\sqcap X$. *A complete lattice* $(L, \sqsubseteq)$ *always has a least element* $\bot_L = \bigsqcup \emptyset$ *and a greatest element* $\top_L = \sqcap \emptyset$.
*Let* $l \in L$*: we define its* upward-closure *as* $\uparrow l = \{l' \mid l' \in L \wedge l \sqsubseteq l'\}$.
*Given a lattice* $L$*, a* basis *is a subset* $B_L \subseteq L$ *such that* $\forall l$, $l = \bigsqcup\{b \in B_L \mid b \sqsubseteq l\}$.
*An element* $l \in L$ *is* completely join-irreducible *if whenever* $l = \bigsqcup X$ *for some* $X \subseteq L$*, then* $l \in X$.

The following lemma is crucial because of the fact that throughout our work we often limit ourselves to finite structures.

**Lemma 2.1.1.** *(**Finite lattices are complete**).*
*Every finite lattice* $(L, \sqsubseteq)$ *is also a complete lattice.*

***Proof***:
Induction on the size of $X$, where $X \subseteq L$.
**Base cases**:

* $|X| = 0$, which implies that $X = \emptyset$. The thesis holds since $\bigsqcup X = \bot_L$. The bottom element exists, because $L$ is a lattice.

**Inductive case**: the induction hypothesis declares that every set $X$ of size $n$ admits a join and a meet. The induction step is carried out by showing that this holds with size $n + 1$ as well.

* $|X| \geq 1$. Let $X = X' \cup \{x\}$, where $|X'| = n$. The inductive hypothesis states that $\bigsqcup X'$ exists. Now, let us see if the equality $\bigsqcup X = (\bigsqcup X') \sqcup x$ holds. Let $z$ be equal to $(\bigsqcup X') \sqcup x$. We want to show that $z$ is an upper bound for $X$. Observe that, for any $y \in X$, we have two cases:

  1. $y \in X'$. The inequalities $X' \sqsubseteq \bigsqcup X' \sqsubseteq z$ hold, then $y \sqsubseteq z$;
  2. $y = x$, then obviously $y \sqsubseteq z$.

  Now, let us see if $z$ is the least upper bound for $X$. For any upper bound $z'$, we have that $\bigsqcup X' \sqsubseteq z'$ and $x \sqsubseteq z'$, since $x \in X$. As a consequence, $z = \bigsqcup X' \sqcup x \sqsubseteq z'$, which is the thesis. □

**Example 2.1.1.** *(**A Powerset Lattice**).*
*The* powerset *of any set $X$, ordered by subset inclusion, engenders the lattice $(2^X, \sqsubseteq)$. The diagram below represents the lattice $(2^X, \sqsubseteq)$, where $X = \{a, b, c\}$. As for the operators, the* union *operator acts as* join*, whereas the* intersection *acts as the dual* meet *operator. The* top *element is $X$, while the* bottom *one is $\emptyset$. The most intuitive way to find a basis is considering the set of singletons $B_{2^X} = \{\{x\} \mid x \in X\}$, which are also* join-irreducible *elements. Observe that any lattice's element can be generated by the union of the basis elements it includes (i.e. set inclusion):*
*e.g.* $\{a, b\} = \{a\} \cup \{b\}$.

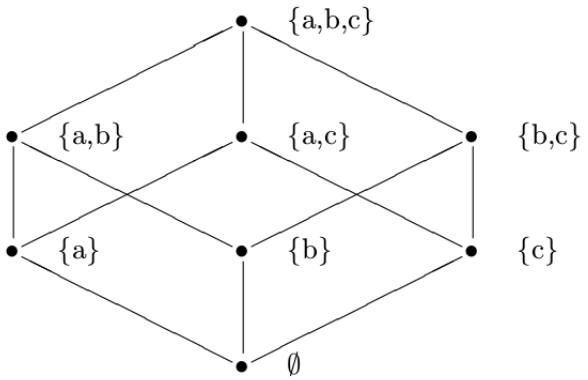



**Figure 2.1:** A powerset lattice [20].

**Definition 2.1.2.** *(**Monotone functions, Least and Greatest fixpoints**).*
*A function $f$: $L \to L$ is* monotone *if $\forall\ l, l'$, if $l \sqsubseteq l'$ then $f(l) \sqsubseteq f(l')$.*
*A consequence of Knaster-Tarski theorem ([8], [24]) is that any monotone function on a complete lattice has* least and greatest fixpoints *($\mu f$ and $\nu f$), which can be computed as the meet of all pre-fixpoints (respectively, the join of all post-fixpoints):*

* $\mu f = \sqcap\{l \mid f(l) \sqsubseteq l\}$;
* $\nu f = \bigsqcup\{l \mid l \sqsubseteq f(l)\}$.

**Theorem 2.1.1.** *(**Kleene's Iteration**).*
*Let $L$ be a lattice and let $f$: $L \to L$ be a monotone function. We know that, in a finite lattice, every chain has finite height. Consider the finite ascending chain $(f^n(\bot))_n$, where $n \in \mathbb{N}$, defined as follows:*

* $f^0(\bot) = \bot$;
* $f^{n+1}(\bot) = f(f^n(\bot))$.

*Let $\lambda_L$ be the height of the given lattice $L$ (i.e. the length of the maximum strictly ascending (finite) chain). Then:*

* $\mu f = f^n(\bot)$ *for some natural* $n \leq \lambda_L$;
* $\nu f = (f^n(\top))_n$ *(finite descending chain).*

**Definition 2.1.3.** *(**Directed Subset**).*
*Given a lattice $L$, a subset $X \subseteq L$ is* directed *if $X \neq \emptyset$ and every pair of elements in $X$ has an upper bound in $X$.*

## 2.2 Tuples

This section provides an overview on tuples of elements, including both their notation and some of their useful properties.

### 2.2.1 Notation

* Given a set $A$, an $n$-tuple belonging to the set $A^n$ will be denoted by a boldface small letter (eg) $\boldsymbol{a}$);
* a tuple will be indexed with indices ranging in the interval $\{1, ..., n\}$, and the $i$-th tuple's item will be denoted as $a_i$;
* an underlined $n \in \mathbb{N}$ ($\underline{n}$) will denote the integer interval $\{1, .., n\}$;
* $\boldsymbol{a}_{i,j}$, for a tuple $\boldsymbol{a} \in A^n$ and two naturals $i, j \in \underline{n}$, is syntactic sugar for representing the subvector $a_i a_{i+1} ... a_j$.

### 2.2.2 Basic properties

**Definition 2.2.1.** *(**Pointwise Order**).*
*Given a lattice $(L, \sqsubseteq)$, we will denote by $(L^n, \sqsubseteq)$ 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 $i \in \underline{n}$.*

**Definition 2.2.2.** *(**Lexicographic Order**).*
*Given a partial order $(P, \sqsubseteq)$ (not necessarily a lattice), the pair $(P^n, \preceq)$ will denote the set of $n$-tuples equipped with the* lexicographic order*, i.e. for $\boldsymbol{l}, \boldsymbol{l'} \in P^n$, we have the following inductive definition:*

$$\boldsymbol{l} \preceq \boldsymbol{l'} \text{ if either } l_n \sqsubset l'_n \text{ or } l_n = l'_n \text{ and } \boldsymbol{l}_{1,n-1} \preceq \boldsymbol{l'}_{1,n-1}$$

*Conversely to the usual lexicographic order for strings, in our definition the most relevant item for the order relation is the last one.*

Observe that if $(L, \sqsubseteq)$ is a lattice, $(L^n, \preceq)$ is a lattice as well.

**Definition 2.2.3.** *(**Truncated Lexicographic Order**).*
*Let $(P, \sqsubseteq)$ be a partial order and let $n \in \mathbb{N}$. For $i \in \underline{n}$ we define a preorder $\preceq_i$ on $P^n$ by letting, for $\boldsymbol{x}, \boldsymbol{y} \in P^n$, $\boldsymbol{x} \preceq_i \boldsymbol{y}$ if $\boldsymbol{x}_{i,n} \preceq \boldsymbol{y}_{i,n}$.*
*Whenever $\sqsubseteq$ is a well-order, given $X \subseteq P^n$, with $X \neq \emptyset$ and $i \in \underline{n}$, the notation*

$$min_{\preceq_i} X$$

*denotes the vector $\boldsymbol{x} = (\bot, ..., \bot, x_i, ..., x_n)$, where $\boldsymbol{x}_{i,n} = min_{\preceq}\{\boldsymbol{l}_{i,n} \mid \boldsymbol{l} \in X\}$.*

## 2.3 Systems of Fixpoint Equations

In this section we introduce the notion of system of fixpoint equations (over some lattice), along with the formal definition of its solution, taken from [2].

**Definition 2.3.1.** *(**System of Equations**).*
*Let* L *be a lattice. A* system of equations E *over* L *is a list of equations of the following form:*

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

*where* $f_i : \mathrm{L}^m \to \mathrm{L}$ *are monotone functions and* $\eta_i \in \{\mu, \nu\}$*, where* $\mu$ *and* $\nu$ *represent, respectively, least and greatest fixpoints. A more compact, vector-like notation for systems is the following:*

$$\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$$

*where* $\boldsymbol{x} = (x_1, ..., x_m)$*,* $\boldsymbol{\eta} = (\eta_1, ..., \eta_m)$ *and* $\boldsymbol{f} = (f_1, ..., f_m)$*.*
$\mathrm{E} = \emptyset$ *is the system without any equations.*

The pre-solutions of a system, introduced below, will be useful in defining the solution of a system.

**Definition 2.3.2.** *(**Pre-solution**).*
*Let* L *be a lattice and let* E *be a system of equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$*. A* pre-solution *of* E *is any tuple* $\boldsymbol{u} \in \mathrm{L}^m$ *such that* $\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{u})$*.*

The vector $\boldsymbol{f}$ can be viewed as a function $\boldsymbol{f} : \mathrm{L}^m \to \mathrm{L}^m$, thus, pre-solutions could also be seen as the fixpoints of $\boldsymbol{f}$. Moreover, each $f_i$ is a monotone function, and $\boldsymbol{f}$ is therefore monotone over $(\mathrm{L}^m, \sqsubseteq)$. In addition, we know from lattice theory that the set of fixpoints of $\boldsymbol{f}$, that is to say, the pre-solutions of the system, is a sublattice.
Systems of equations can be solved thanks to the usual substitution method, whose formalisation and adaptation for the case we are studying is provided below.

**Definition 2.3.3.** *(**Substitution**).*
*Given a system* E *of* $m$ *equations over a lattice* L *of the kind* $\boldsymbol{x} =_{\eta} \boldsymbol{f}(\boldsymbol{x})$*, an index* $i \in \underline{m}$ *and an element* $l \in \mathrm{L}$*, we write* $\mathrm{E}[x_i := l]$ *for the system of* $m - 1$ *equations obtained from* E *by removing the i-th equation and replacing* $x_i$ *by* $l$ *in the other equations, i.e. if* $\boldsymbol{x} = \boldsymbol{x'} x_i \boldsymbol{x''}$*,* $\boldsymbol{\eta} = \boldsymbol{\eta'} \eta_i \boldsymbol{\eta''}$ *and* $\boldsymbol{f} = \boldsymbol{f'} f_i \boldsymbol{f''}$*, then* $\mathrm{E}[x_i := l]$ *is equal to:*

$$\boldsymbol{x'}\boldsymbol{x''} =_{\boldsymbol{\eta'},\boldsymbol{\eta''}} \boldsymbol{f'}\boldsymbol{f''}(\boldsymbol{x'}, l, \boldsymbol{x''})$$

Let $f[x_i := l] : \mathrm{L}^{m-1} \to \mathrm{L}$ be defined by:

$$f[x_i := l](\boldsymbol{x'}, \boldsymbol{x''}) = f(\boldsymbol{x'}, l, \boldsymbol{x''})$$

Then, the system $\mathrm{E}[x_i := l]$ has $m - 1$ equations:

$$x_j =_{\eta_j} f_j[x_i := l](\boldsymbol{x'}, \boldsymbol{x''})\ j \in \{i, ..., i-1, i+1, ...n\}$$

The previous definition is exploited to define solutions in a recursive manner.

**Definition 2.3.4.** *(**Solution**).*
*Let* L *be a lattice and let* E *be a system of m equations on* L *of the kind* $\boldsymbol{x} =_\eta \boldsymbol{f}(\boldsymbol{x})$. *The* solution *of* E, *denoted* $sol(\mathrm{E}) \in \mathrm{L}^m$, *is defined recursively as follows:*

$$sol(\emptyset) = ();$$
$$sol(\mathrm{E}) = (sol(\mathrm{E}[x_m := u_m]), u_m) \text{ where } u_m = \eta_m(\lambda x.f_m(sol(\mathrm{E}[x_m := x]), x)).$$

*Being vectors, solutions are endowed with indices, thus* $sol_i(\mathrm{E})$ *represents the i-th component of the solution.*

Intuitively, the final solution is obtained by considering the last variable as a fixed parameter $x$ and solving the remaining system of $m-1$ equations recursively. The following $(m-1)$-tuple (parametric on $x$) is therefore produced:

$$\boldsymbol{u}_{1,m-1}(x) = sol(\mathrm{E}[x_m := x])$$

Such $x$-parametrised solution is then grafted into the last equation, which becomes a single-variable equation of the following shape:

$$x =_{\eta_m} f_m(\boldsymbol{u}_{1,m-1}(x), x)$$

which can be solved by taking the least or greatest fixpoint (depending on $\eta_m$) for the function $\lambda x.f_m(\boldsymbol{u}_{1,m-1}(x), x)$. Finally, this procedure will yield the $m$-th component $u_m$ of the solution, which is in turn inserted into the parametric solution $\boldsymbol{u}_{1,m-1}(x)$ previously computed.
The following lemma ensures that the definition of solution given above is well-given, by observing that we are dealing with monotone functions.

**Lemma 2.3.1.** *(**Solution is monotone**).*
*Let* E *be a system of* $m > 0$ *equations of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *over a lattice* L. *For* $i \in \underline{m}$, *the function* $g : \mathrm{L} \to \mathrm{L}^{m-1}$, *defined by* $g(x) = sol(\mathrm{E}[x_i := x])$, *is monotone.*

**Lemma 2.3.2.** *(**The Solution is a Pre-solution**).*
*Let* E *be a system of m equations of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *over a lattice* L *and let* $\boldsymbol{u}$ *be its solution. Then* $\boldsymbol{u}$ *is a pre-solution, that is to say, the equality* $\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{u})$ *holds.*

**Proof.** Let us proceed by induction on $m$.

* Base case ($m = 0$): it holds trivially.
* Inductive case ($m > 0$): let $\boldsymbol{u} = \boldsymbol{u'}u_m$, $f = \boldsymbol{f'}f_m$ and $\boldsymbol{x} = \boldsymbol{x'}x_m$. Since $\boldsymbol{u'} = sol(\mathrm{E}[x_m := u_m])$, thanks to the inductive hypothesis, the following

expression is true:

$$\boldsymbol{u'} = \boldsymbol{f'}[x_m := u_m](\boldsymbol{u'}) = \boldsymbol{f'}(\boldsymbol{u}) \quad .$$

Now, by the definition of solution, we have that

$$u_m = \eta_m(\lambda x.f_m(sol(\mathrm{E}[x_m := x]), x))$$

Hence $u_m = f_m(sol(\mathrm{E}[x_m := u_m]), u_m))$. Since $sol(\mathrm{E}[x_m := u_m]) = \boldsymbol{u'}$, we infer that $u_m = f_m(\boldsymbol{u'}, u_m) = f_m(\boldsymbol{u})$, and thus $\boldsymbol{u} = \boldsymbol{f}(\boldsymbol{u})$, which is the thesis.

□

### 2.3.1 Examples

We will now consider some small-sized ($m = 2$) systems of fixpoint equations, intuitively compute their solutions, and point out what makes them different from the systems of equations with plain (i.e. with no least/greatest fixpoints) equalities anyone is familiar with. Observe that our examples will contain both least and greatest fixpoint equations.

Consider the following system of equations on a powerset lattice $\mathcal{P}(A)$, where $A$ is a fixed set:

$$\begin{cases} x =_\mu x \cap y \\ y =_\nu x \cup y \end{cases}$$

By way of a trivial substitution, you can see that the solution is equal to:

$$\{x = \emptyset, y = A\},$$

because $\emptyset$ is clearly the smallest (with respect to set inclusion) solution for the first equation, whereas $A$ is the greatest solution for the second one.

Clearly, the order in which equations are written could affect the final solution. Let us invert the order of the previous system's equations.

$$\begin{cases} y =_\nu x \cup y \\ x =_\mu x \cap y \end{cases}$$

In this case nothing changes, since the solution remains equal to $\{x = \emptyset, y = A\}$. Now we try to swap the set operators of the first system and see what happens to the solution:

$$\begin{cases} x =_\mu x \cup y \\ y =_\nu x \cap y \end{cases}$$

Here the solution is:

$$\{x = A, y = A\},$$

which can be easily obtained by putting $x$ equal to $y$ in the first equation (we remind that the union between a set $S$ and itself is obviouly smaller than the union between $S$ and any $S'$ such that $S \neq S'$).
Now we swap the equations of the above system:

$$\begin{cases} y =_\nu x \cap y \\ x =_\mu x \cup x \end{cases}$$

We finally managed to attain a different solution:

$$\{x = \emptyset, y = \emptyset\}$$

We simply put $x = y$ and then substituted in the second equation.
Note that the intersection between a set $S$ and itself is obviously greater than the intersection between $S$ and any $S'$ such that $S \neq S'$.

It can be easily seen that such a strong dependency on the order of equations could occur only when the system contains a mix of least and greatest fixpoint equations, an event which happens very often, for instance when the system is generated from the translation of a $\mu$-calculus formula.

## 2.4 The $\mu$-calculus: a brief introduction

Modal $\mu$-calculus ($L_\mu$) is a logic that extends *propositional modal logic* by means of the addition of inductive definitions. This new feature leads naturally to the definition of a *least fixed point operator* ($\mu$) and a *greatest fixed point operator* ($\nu$), which together make $\mu$-calculus a *fixed-point logic*. This additional (with respect to propositional modal logic) machinery provides $\mu$-calculus both with greater expressive power and lesser ease of comprehension. But even more importantly, this logic allows to highlight the connections between modal and temporal logics, automata theory and game theory.
The $\mu$-calculus was conceived by Dana Scott and Jaco de Bakker, and was further developed and improved by Dexter Kozen (1983) into its more popular modern version. Nowadays, it is mainly used in the description of *labelled transition systems'* properties and, accordingly, for their verification.

### 2.4.1 An eminent precursor: Hennessy-Milner Logic (HML)

As regards older logics in the field of process calculi, we could state that one of the most influential precursors of $\mu$-calculus has been introduced in the late 70s by Hennessy and Milner, who were also the first to leverage labelled *Kripke structures* ([16], also known as labelled transition systems) as a raw model of concurrent behaviour. Hennessy and Milner devised a primitive modal logic where modalities refer to actions. More formally, the set $\mathcal{M}$ of Hennessy-Milner formulae over a set of actions `Act` is given by the following abstract syntax (see [1]):

$$\phi, \psi ::= tt \mid ff \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle a \rangle \phi \mid [a]\phi$$

where $a \in$ `Act`, while $tt$ and $ff$ denote 'true' and 'false', respectively.
The meaning of a $\mathcal{M}$ formula is given in terms of the collection of processes that satisfy such formula, and it can be briskly grasped as follows:

- all processes satisfy $tt$;
- no process satisfies $ff$;
- a process satisfies $\phi \wedge \psi$ if and only if it satisfies both $\phi$ and $\psi$;
- a process satisfies $\phi \vee \psi$ if and only if it satisfies either $\phi$ or $\psi$;
- a process satisfies $\langle a \rangle \phi$ for some $a \in$ `Act` if and only if it affords an $a$-labelled transition leading to a state satisfying $\phi$;
- a process satisfies $[a]\phi$ for some $a \in$ `Act` if and only if all of its $a$-labelled transitions lead to a state satisfying $\phi$.

But nonetheless, HML suffers from a lack of expressive power, since it has no means of formalising temporal connectives such as *always* and *eventually*, whose formalisation will indeed be the strong point of $\mu$-calculus. More generally, HML cannot express properties of unbounded or infinite computations, that are commonly of interest (think, e.g., of deadlock freedom or liveness).

### 2.4.2 The syntax of modal $\mu$-calculus

The main feature of $\mu$-calculus is a clever use of fixpoint operators, and therein lies its expressive strength. We will not delve too much into the deep meaning of the use of fixpoint operators, because it would require a overly theoretical excursus. The language's syntax is summarised below (see [4] for more details).

**Definition 2.4.1.** *(**The syntax of $\mu$-calculus**).*
*Consider the following disjoint sets:*

* *Var: the set of propositional variables* $(x, y, z, ...)$*;*
* *Prop: the set of propositional symbols* $(p, q, r, ...)$*.*

*$\mu$-calculus formulae are defined inductively as:*

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

*where* $\eta \in \{\mu, \nu\}$*.*

*Unlabelled Transition Systems* (also called *Kripke structures*) play a fundamental role for the definition of $\mu$-calculus' (and many other calculi) semantics, since the semantics of a formula is given with respect to such systems.

**Kripke structure**
A Kripke structure over $Prop$ is a 4-tuple $K = (\mathbb{S}, I, R, L)$ consisting of:

* a finite set of states $\mathbb{S}$;
* a set of initial states $I \subseteq \mathbb{S}$;
* a transition relation $R \subseteq \mathbb{S} \times \mathbb{S}$ ($R$ is often denoted as $\rightarrow$);
* a labelling (or *interpretation*) function $L : \mathbb{S} \rightarrow 2^{Prop}$.

Now, let $\phi$ and $\rho : Prop \cup PVar \rightarrow 2^{\mathbb{S}}$ be, respectively, a $\mu$-calculus formula and a function (*environment*) which maps each proposition or propositional variable to the subset of $\mathbb{S}$ where such proposition or propositional variable is true.
$[\![\phi]\!]_\rho$ indicates the semantics of $\phi$, defined in the usual way ([5]).

**Semantics of $L_\mu$**
Given a Kripke structure $K$ (i.e. the 4-tuple explained earlier) and an environment $\rho : Prop \cup PVar \rightarrow 2^{\mathbb{S}}$, the semantics of a $L_\mu$ formula $\phi$ (written $[\![\phi]\!]_\rho$) is defined as follows:

* $[\![p]\!]_\rho = \rho(p)$
* $[\![x]\!]_\rho = \rho(x)$
* $[\![\phi_1 \wedge \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cap [\![\phi_2]\!]_\rho$

* $[\![\phi_1 \vee \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cup [\![\phi_2]\!]_\rho$
* $[\![\Box\phi]\!]_\rho = \{s \mid \forall t.s \to t \implies t \in [\![\phi]\!]_\rho\}$
* $[\![\Diamond\phi]\!]_\rho = \{s \mid \exists t.s \to t \implies t \in [\![\phi]\!]_\rho\}$
* $[\![\nu x.\phi]\!]_\rho = \bigcup\{S \subseteq \mathbb{S} \mid S \subseteq [\![\phi]\!]_{\rho[x:=S]}\}$
* $[\![\mu x.\phi]\!]_\rho = \bigcap\{S \subseteq \mathbb{S} \mid S \supseteq [\![\phi]\!]_{\rho[x:=S]}\}$

**Definition 2.4.2.** *(**Alternation Depth of Formulae** [6]).*
*Let $\phi$ be a μ-calculus formula in* Positive Normal Form*, i.e. a negation-free formula in which no variable is bound more than once. Then the* alternation depth*, $ad(\phi)$, of $\phi$ is defined inductively as follows:*

* *if $\phi$ contains closed top-level fixpoint-subformulae $\gamma_1, ..., \gamma_n$ then:*

$$ad(\phi) = max(ad(\phi'), ad(\gamma_1), ..., ad(\gamma_n))$$

  *where $\phi'$ is obtained from $\phi$ by substituting new atomic proposisions $a_1, ..., a_n$ for $\gamma_1, ..., \gamma_n$;*

* *if $\phi$ contains no closed top-level fixpoint-subformulae then $ad(\phi)$ is defined as follows:*
  - *$ad(a) = ad(x) = 0$, for any atomic proposition $a$ and variable $x$;*
  - *$ad(\phi_1 \wedge \phi_2) = ad(\phi_1 \vee \phi_2) = max(ad(\phi_1), ad(\phi_2))$;*
  - *$ad(\Box\phi) = ad(\Diamond\phi) = ad(\phi)$;*
  - *let $\eta \in \{\mu, \nu\}$, and let $\overline{\eta}$ be the dual of $\eta$. Then:*

$$ad(\eta x.\phi) = max(1, ad(\phi), 1 + ad(\overline{\eta}x_1.\phi_1), ..., 1 + ad(\overline{\eta}x_n.\phi_n))$$

    *where $\overline{\eta}x_1.\phi_1, ..., \overline{\eta}x_n.\phi_n$ are top-level $\overline{\eta}$-subformulae of $\phi$.*

**Example 2.4.1.** *For $\phi = \nu x_1.\mu x_2.(x_1 \vee x_2 \vee \nu y_1.\mu y_2.\nu y_3.(y_1 \wedge y_2 \wedge y_3))$ we obtain $ad(\phi) = 3$.*

**Example 2.4.2.** *Let us consider the formula $\phi = \nu x_2.((\mu x_1.(p \vee \Diamond x_1)) \wedge \Box x_2)$, which requires that* from all reachable states, there exists a path that eventually reaches a state where p holds*. The equational form of $\phi$ is:*

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

*Consider the transition system (depicted in the next page) $(\mathbb{S}, \to)$, where $\mathbb{S} = \{a, b\}$, with $p$ that holds only on state $b$. In order to apply the equations on the powerset lattice $2^{\mathbb{S}}$, the system should be written in a way that takes into account the meaning*

*of boolean operators and propositional variables on such lattice:*

$$\begin{cases} x_1 =_\mu \{b\} \cup \Diamond x_1 \\ x_2 =_\nu x_1 \cap \Box x_2 \end{cases}$$

*The normalised version of the previous system, that is to say, an equivalent system with a single operator for each equation, is shown below:*

$$\begin{cases} x_1 =_\mu p \vee x_3 \\ x_3 =_\mu \Diamond x_1 \\ x_2 =_\nu x_1 \wedge x_4 \\ x_4 =_\nu \Box x_2 \end{cases}$$

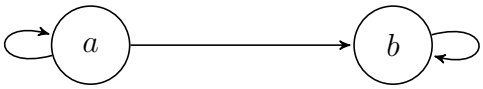



**Figure 2.2:** A transition system.

## 2.5 Parity Games

A *Parity game* is played on a particular kind of graph, called *parity graph*, which is made up of a directed graph ($V$, $E$) and a priority function $p$: $V \to [d]$, where $d$ is a natural number. A formal definition of parity game follows (see [13] for more details).

**Definition 2.5.1.** *(**Parity game**).*
*A parity game is a tuple* $\Gamma = (V, E, p, (V_\Diamond, V_\Box))$*, where* $G = (V, E, p)$ *is the* game graph *of* $\Gamma$ *and* $V$ *is partitioned into two sets* ($V_\Diamond$, $V_\Box$).
*The two players, called* $\Diamond$ *and* $\Box$ *(or* $\exists$ *and* $\forall$ *in the next chapter) form a finite or infinite path in the game graph by passing a token to each other along the edges. First and foremost, an initial vertex is chosen and the token is placed on it. During the game, the vertex such token is placed on gives rise to two simple rules:*

- *let the token be on a vertex in the* $V_\Diamond$ *set, then player* $\Diamond$ *moves the token along one of the outgoing edges with respect to the current vertex;*
- *conversely, player* $\Box$ *should act similarly on a* $V_\Box$ *vertex.*

As any other game, parity games are equipped with a formal *winning condition* which changes according to the game's finiteness.

**Definition 2.5.2.** *(**Winning conditions**).*
*As stated earlier, finite and infinite games lead to distinct requirements.*

- Finite play*: the winner of a finite play is the player whose opponent is unable to move.*
- Infinite play*: let* $\pi = (v_1, v_2, ...)$ *be a play (that is to say, an infinite path covered by players on the game graph). Let* $\mathrm{Inf}(\pi)$ *denote the set of priorities appearing infinitely often in the nodes covered by the play. We say that* $\pi$ *is a* winning play *for* $\Diamond$ *if* $\min(\mathrm{Inf}(\pi))$ *is even. On the contrary,* $\pi$ *is winning for* $\Box$ *if* $\min(\mathrm{Inf}(\pi))$ *is odd.*

**Definition 2.5.3.** *(**Strategy, Consistent play, Winning strategy**).*
*A function* $\sigma : V_\Diamond \to V$ *is a* strategy *for player* $\Diamond$ *if* $(v, \sigma(v)) \in E \; \forall v \in V_\Diamond$.
*A play* $\pi = (v_1, v_2, ...)$ *is* consistent *with a strategy* $\sigma$ *(for player* $\Diamond$*) in the event that* $v_{l+1} = \sigma(v_l)$, $\forall l \in \mathbb{N}$.
$\sigma$ *is a* winning strategy *for player* $\Diamond$ *(*$\Box$*) with respect to set* $W \subseteq V$*, if every play (consistent with* $\sigma$*) starting from a vertex in* $W$ *is winning for player* $\Diamond$ *(*$\Box$*).*

The problem of *solving* or *deciding* a parity game consists in (given a parity graph and a vertex to start the game from) deciding whether player $\Diamond$ ($\Box$) has a winning strategy from such vertex. The large amount of work done in order to determine the complexity of parity game solving is justified by the fact that the problem is *polynomial time equivalent* to the modal $\mu$-calculus model checking. Furthermore, the problem's importance derives also from a complexity theory perspective: in fact, it has been shown that this problem is in **NP** and **Co-NP** complexity classes, or better yet, **UP** (*unambiguous non-deterministic polynomial-time*) and **Co-UP**,

as well as in **QP** (*quasipolynomial time*) class. Anyway, it is not known whether parity games can be solved in polynomial time (which is an unsolved problem in computer science).

A remarkable fact about parity games is that any parity game admits a convenient translation into a system of least and greatest fixpoint equations over the boolean lattice (see [14], [15] and [18]), that is, the lattice $\{false, true\}$, with ordering $false \sqsubseteq true$.

**Example 2.5.1.** *In the parity game represented below, each node has been coloured by a natural number constituting its priority. Two* winning regions *can be found out, one for player* 0*, whereas the other for player* 1*. This means that if the game starts from any node contained in the so called "player* 0*'s winning region", then we can be sure that player* 0 *eventually wins (the conditions for player* 1 *are dual). Edges with a diamond-shaped white head lead more fastly to an infinite play where it is easy to check the winning condition for player* 0*, while edges with black, diamond-shaped heads have the same purpose for the other player.*

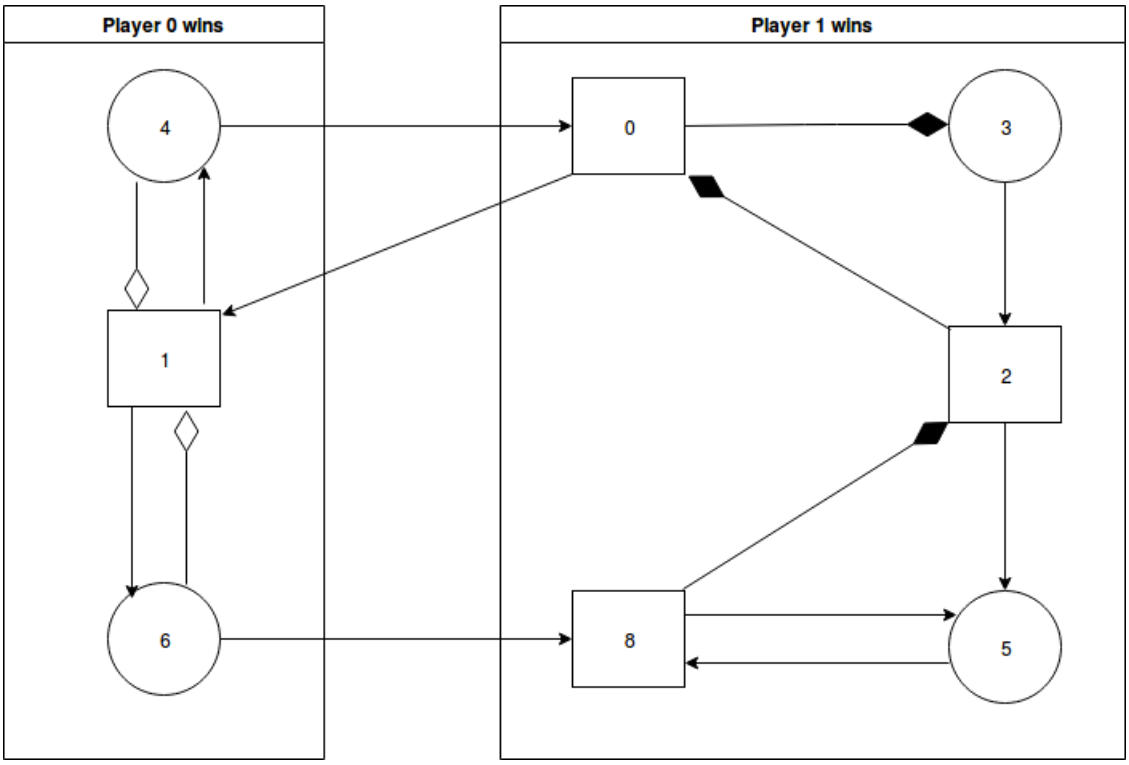



**Figure 2.3:** A parity game. The circles belong to player 0, the squares to player 1.

The game shown in the example above can be easily converted into a system of fixpoint boolean equations by way of the following straightforward recipe:

- introduce a variable for each node, with an index equal to that node's priority;
- 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;

* the right-hand side of an equation should include all the variables corresponding to all the successors of the node related to the left-hand variable;

* as regards the boolean operators between variables, we should first consider that we could be interested either in player 0's or in player 1's final win. This observation leads to two cases:

  – if we want to know whether player 0 wins, then the right-hand side of each equation with a player 0's variable as its left-hand side should be put under a $\bigvee$ operator, while a $\bigwedge$ operator should be used for the remaining equations;

  – otherwise, the right-hand side of each equation with a player 1's variable as its left-hand side should be put under an $\bigvee$ operator, while a $\bigwedge$ operator should be used for the remaining equations.

Thus, the outcome of this recipe applied to the game introduced in Example 2.5.1 is:

$$\begin{cases} x_0 =_\nu x_1 \wedge x_3 \\ x_1 =_\mu x_4 \wedge x_6 \\ x_2 =_\nu x_0 \wedge x_5 \\ x_3 =_\mu x_2 \\ x_4 =_\nu x_0 \vee x_1 \\ x_5 =_\mu x_8 \\ x_6 =_\nu x_1 \vee x_8 \\ x_8 =_\nu x_2 \wedge x_5 \end{cases}$$

Let us see if the solution of the system obtained by means of our recipe is consistent with the one inferred by the parity game of the example. Looking at the winning regions in the game graph, we deduce (assuming that we want to answer the question *does player* 0 *win?*) the assignments $x_0 = false, x_1 = true$ (because node 1 is in the winning region for player 0), $x_2 = false, x_3 = false, x_4 = true$ (because node 1 belongs to the winning region for player 0), $x_5 = false, x_6 = true, x_8 = true$.
Such assignments verify each equation when substituted into the system.
Note that this equivalence between parity games and system of boolean equations will allow us to use our tool both as an equational system solver and a parity game solver.

### 2.5.1 Progress Measures

*Progress Measures* constitute a technique that associates to each vertex of the parity graph a monotonically increasing *measure*. The measure of each vertex is *lifted* based on the measures of its successors. As a matter of fact, the measure represents a statistic of the most optimal play so far from the vertex, without storing the full plays explicitly. Basically, Progress Measures account for how favourable the game is for a given player.
Although the next chapter includes a detailed study about Progress Measures in the case of fixpoint games, below we report some definitions and results concerning

this topic provided in [13], which addresses this technique in the field of general parity games.

**Definition 2.5.4.** *(**Parity Progress Measure**).*
*Let $G = (V, E, p : V \to [d])$ be a parity graph. A function $\rho : V \to \mathbb{N}^d$ is a* Parity Progress Measure *for $G$ if for all $(v, w) \in E$, we have $\rho(v) \geq_{p(v)} \rho(w)$, and the inequality is strict if $p(v)$ is odd.*

Now, if $G = (V, E, p : V \to [d])$ is a parity graph, then for every $i \in [d]$, we write $V_i$ to denote the set $p^{-1}(i)$ of vertices with priority $i$ in parity graph $G$. Let $n_i = |V_i|$, $\forall i \in [d]$. Thus, let $M_G$ be the following finite subset of $\mathbb{N}^d$:

* $M_G = [1] \times [n_1 + 1] \times [1] \times [n_3 + 1] \times ... \times [1] \times [n_{d-1} + 1]$, if $d$ is even;
* $M_G = [1] \times [n_1 + 1] \times [1] \times [n_3 + 1] \times ... \times [n_{d-2} + 1] \times [1]$, if $d$ is odd.

That is to say, $M_G$ is the finite set of $d$-tuples of integers with only zeroes on even positions, and non-negative integers bounded by $|V_i|$ on every odd position $i$.
Let $\Gamma = (V, E, p, (V_\Diamond, V_\Box))$ be a parity game and let $G = (V, E, p)$ be its game graph.
Let $M_{G^\top}$ be equal to the set $M_G \cup \{\top\}$, where $\top$ is an extra element.
The notation $Prog(\rho, v, w)$ denotes the least $m \in M_G^\top$ such that:

* $m \geq_{p(v)} \rho(w)$, if $p(v)$ is even;
* $m \geq_{p(v)} \rho(w)$, or $m = \rho(w) = \top$, if $p(v)$ is odd,

where $\geq$ represents the lexicographic order on $M_G$ with $\top$ as the biggest element.

**Definition 2.5.5.** *(**Game Parity Progress Measure**).*
*A function $\rho : V \to M_G^\top$ is a* Game Parity Progress Measure *if $\forall v \in V$, we have that:*

* *if $v \in V_\Diamond$, then $\rho \geq_{p(v)} Prog(\rho, v, w)$ for some pair $(v, w) \in E$;*
* *if $v \in V_\Box$, then $\rho \geq_{p(v)} Prog(\rho, v, w)$ for all pairs $(v, w) \in E$.*

*Notation: $\|\rho\|$ denotes the set $\{v \in V \mid \rho(v) \neq \top\}$.*

**Definition 2.5.6.** *(**Strategies as Game Progress Measures**).*
*Let $\rho$ be a Game Progress Measure. A* strategy *$\tilde{\rho} : V_\Diamond \to V$ for player $\Diamond$ is a function that outputs a successor $w$ of $v$ such that $\rho(w)$ is minimised.*

**Theorem 2.5.1.** *If $\rho$ is a Game Parity Progress Measure, then $\tilde{\rho}$ is a winning strategy for player $\Diamond$ from $\|\rho\|$.*

**Theorem 2.5.2.** *There is a Game Progress Measure $\rho : V \to M_G^\top$ such that $\|\rho\|$ is the winning set of player $\Diamond$.*

See [13] for all the proofs.

### 2.5.2 Parity Game Solvers

There are mainly two categories of parity game solvers. The algorithms belonging to the first category perform local updates in an iterative way, until a fixpoint is reached. Each vertex is equipped with some measure which records the best game that can be played from that vertex from either player so far. Thus, by updating measures based on successor vertices, the game is played backwards. The final measures indicate the winning player of each vertex and a *winning strategy* for one or both players. The *Strategy Improvement* and the *Progress Measures* algorithms belong to this first category.
Other algorithms use the *attractor* computation to partition the game into regions that share a certain property. The *recursive Zielonka* ([26]) algorithm and the *priority promotion* ([3]) algorithms fall into this category. A comprehensive overview of this topic can be found in [30].

CHAPTER 3

# Fixpoint Games

This chapter constitutes the core of the thesis: indeed, it includes a mostly theoretical, detailed presentation of all the concepts we need in order to design our solver. In particular, fixpoint games and Progress Measures are dealt with in an extensive manner, also providing examples that help the reader to realise even the hardest theoretical sections. We decided to keep the number of proofs to a minimum, since some of the most complex ones would be more suitable for a fully theoretical dissertation.

## 3.1 The Game

The definition of fixpoint game constitutes the starting point of the game-theoretic approach used in this chapter. The connection between the game presented in this section and the model checking problem is given by the fact that the game characterisation allows to verify whether an element of the basis is smaller than the solution (fixpoint) of a certain equation (given two lattice elements $x$ and $y$, we say that $x$ is smaller than $y$ if the relation $x \sqsubseteq y$ holds). The similarity with model checking is obvious, because when we model-check $\phi$, a certain formula of the $\mu$-calculus (on a given Kripke structure) we are required to see if $s_0$, the initial state of the transition system, is contained in the solution of the system of equations derived from $\phi$.

**Definition 3.1.1.** *(**Fixpoint Game**).*
*Let* L *be a complete lattice and let* $B_L$ *be a basis of* L *that does not include the* $\bot$ *element. A system* E *of* m *equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *is given.*
*The* fixpoint game *associated with the system is a* parity game*, with two players:*

- *the existential player,* $\exists$*, with* ***positions*** $(b, i)$ *where* $b \in B_L$ *and* $i \in \underline{m}$*, and possible* ***moves*** *defined as* $\mathbf{E}(b, i) = \{\boldsymbol{l} \mid \boldsymbol{l} \in \mathrm{L}^m \wedge b \sqsubseteq f_i(\boldsymbol{l})\}$*;*
- *the universal player,* $\forall$*, with tuples* $\boldsymbol{l} \in \mathrm{L}^m$ *as* ***positions*** *and possible* ***moves*** *defined as* $\mathbf{A}(\boldsymbol{l}) = \{(b, i) \mid i \in \underline{m} \wedge b \in B_L \wedge b \sqsubseteq l_i\}$*.*

In a finite play, the winner is the player whose opponent is unable to move, whereas

in an infinite play the condition can be explained as follows. Let $h$ be the highest index that occurs infinitely often in a pair (position) $(b, i)$. There are two cases:

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

**Theorem 3.1.1.** ***(Correctness and Completeness).***
*Given a system of m equations* E *over a finite lattice* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *with solution* $\boldsymbol{u}$*, then for all* $b \in B_L$ *and* $i \in m$ *the following statement holds:*

$$b \sqsubseteq u_i \qquad \textit{iff} \qquad \exists \textit{ has a winning strategy from position (b,i).}$$

***Proof:***
The complete proof has been omitted; however, some intuition about it is given by restricting the theorem's requirements to a single equation. Since there is a single equation, indices will not be considered throughout the proof.
**Completeness**: if $b \sqsubseteq u$ then $\exists$ has a winning strategy from b.
**Correctness**: if $b \not\sqsubseteq u$ then $\forall$ has a winning strategy from b.

(Case $\eta = \mu$)
Completeness: the given equation is $x =_\mu f(x)$, thus the solution is obviously $u = lfp(f)$. Knaster-Tarski entails $u = f^n(\bot)$, so the hypothesis can be written as $b \sqsubseteq f^n(\bot)$. According to the rules, the existential player can play any $l$ such that $b \sqsubseteq f(l)$. Now, we know that $\bot \notin B_L$ and thus that $n > 0$ (because $n = 0$ would imply $b \sqsubseteq f^0(\bot) = \bot$, which would cause a contradiction since $\bot \notin B_L$).
The fact that $n$ is positive allows us to proceed by induction on the length of the usual chain $(f^n(\bot))_{n\in\mathbb{N}}$.
**Induction hypothesis**: if $b \sqsubseteq f^{n-1}(\bot)$ then $\exists$ has a winning strategy from $b$.
The main idea can be stated as follows: descend the chain $(f^n(\bot))_{n\in\mathbb{N}}$ until the universal player can no longer answer with a move. The sequence of moves will be something like the one below:

* $\exists$ plays $l = f^{n-1}(\bot)$ (from the hypothesis, $b \sqsubseteq f(f^{n-1}(\bot)) = f^n(\bot)$);
* $\forall$ answers with a $b'$ such that $b' \sqsubseteq l = f^{n-1}(\bot)$;
* $\exists$ responds in turn with $l = f^{n-2}(\bot)$ such that $b' \sqsubseteq f(l) = f(f^{n-2}(\bot)) = f^{n-1}(\bot)$. Here the premise is $b' \sqsubseteq f^{n-1}(\bot)$, therefore the induction hypothesis can be applied, from which it follows that $\exists$ has a winning strategy from $b'$ (and from $b$ as well since the game goes from $b$ to $b'$ by means of a move). Hence the thesis.

Correctness: since $u = f^n(\bot)$, the premise becomes $b \not\sqsubseteq f^n(\bot)$. The proof is split into a couple of distinct cases:

1. $b \not\sqsubseteq f^n(\bot)$ could mean that $\exists$ has no moves, therefore $\forall$ clearly wins;

2. given any move by $\exists$, that is, an $l$ such that $b \sqsubseteq f(l)$, we show that there must be a $b' \sqsubseteq l$ such that $b' \not\sqsubseteq u$. The relation $b' \not\sqsubseteq u$ is needed in order to proceed with the game in an inductive fashion. The fact that $b' \not\sqsubseteq u$ is proven by means of a contradiction: if (given an $l$ such that $b \sqsubseteq f(l)$) there existed a $b' \sqsubseteq l$ such that $b' \not\sqsubseteq u$, then we would have had: $l = \bigsqcup\{b' \mid b' \sqsubseteq l\} \sqsubseteq u$, that is, $l \sqsubseteq u$. This would imply the following chain of relations: $b \sqsubseteq f(l) \sqsubseteq f(u) = u$ (because $u$ is a fixpoint), which cannot be true since the premise is $b \not\sqsubseteq u$.

(Case $\eta = \nu$)
Completeness: here the given equation is $x =_\nu f(x)$, thus the solution is of the kind $u = gfp(f)$. In this case Knaster-Tarski entails $u = f^n(\top)$, and therefore the premise becomes $b \sqsubseteq f^n(\top)$. The game can be described as follows:

* $\exists$ plays an $l$ equal to $u$ that satisfies the chain $b \sqsubseteq f(l) = f(u) = u$ (since u is a fixed point);

* $\forall$ answers with $b' \sqsubseteq l = u$, which implies $b' \sqsubseteq u$.

This means that a move has been made and it brings to the original premise; the game therefore proceeds inductively with $\exists$'s win.

Correctness: we know that $u = f^n(\top)$ and thus $b \not\sqsubseteq f^n(\top)$. Observe that $n > 0$, otherwise we would have the following inconsistent chain of relations: $b \not\sqsubseteq u = f^0(\top) = \top \Rightarrow b \not\sqsubseteq \top$, which is obviously false, since every lattice element should be below the top element. So, we end up with two cases:

1. $\exists$ has no moves, thus $\forall$ clearly wins;

2. analogous to the $\eta = \mu$ case.

□

### 3.1.1 A fixpoint game on a single equation

Consider the function $f : 2^A \to 2^A$ (where $A = \{0, 1, 2\}$), defined as follows:

$$f(X) = X \cup \{0\} \cup \{2 \text{ if } 0 \in X\}$$

Its least fixpoint (*lfp*) is equal to the set $\{0, 2\}$, which can be obtained applying *Kleene's iteration*, assuming $\bot = \emptyset$:

* $f(\emptyset) = \{0\}$;

* $f(\{0\}) = \{0, 2\}$;

* $f(\{0, 2\}) = \{0, 2\}$.

The equation we want to play on is therefore $x =_\mu f(x)$, and the lattice's basis we choose is $B_A = \{\{0\}, \{1\}, \{2\}\}$, that is, the one made up of join-irreducible elements.
Now, take a $b \in B_A$ such that $b \subseteq lfp(f)$: we choose $b = \{2\}$. Then, player $\exists$ should

have a winning strategy if the fixpoint game begins at $b$, since the game has been proven to be *complete* in the previous section. The moves of the game are shown here:

1. $\exists$ is at position $b = \{2\}$. It plays an $l \in 2^A$ such that $b \subseteq f(l)$. Such $l$ could be equal to $\{0\}$, because $b \subseteq f(\{0\})$;
2. now, player $\forall$ is at position $\{0\}$ and answers with a $b' \subseteq \{0\}$. Such $b'$ could be equal to $\{0\}$, since $\{0\} \subseteq \{0\}$;
3. at this point, $\exists$ finds itself at $\{0\}$, and replies to $\forall$ with $l = \emptyset$, since $b' \subseteq f(\emptyset)$;
4. thus, $\forall$ cannot make a move from $\emptyset$, because the only $b''$ such that $b'' \subseteq \emptyset$ is $\emptyset$ itself, but $\emptyset \notin B_A$ by definition of basis.

From the moves listed above, we understand that $\exists$ wins the (finite) game because $\forall$ ends up in a *stuck* position. We remark on the fact that player $\exists$ chooses its moves by descending $f^n$, the chain induced by Kleene's procedure: it can be shown that this is the best way to arrive earlier at a winning state (the completeness property has been shown using a similar idea).

We now try to verify if the *correctness* property holds on the same equation. Consider an element of the basis $B_A$ such that $b \not\subseteq lfp(f)$: the choice is $b = \{1\}$, because $\{1\} \notin \{0, 2\}$. The correctness property says that player $\forall$ has a winning strategy whenever the game starts from $b$. The final win of the universal player is showcased below:

* $\exists$ is at position $\{1\}$ and plays $l = \{1, 2\}$, because $\{1\} \subseteq f(l) = \{0, 1, 2\}$;
* so, $\forall$ is in $l = \{1, 2\}$. The rules of the game force the universal player to select a $b \subseteq \{1, 2\}$: let it select $b = \{1\}$;
* the previous choice leads to an infinite game, because $\exists$ ends up again at position $b = \{1\}$.

The fact that the game is infinite, along with the information about the required fixpoint ($\eta = \mu$) let us infer that eventually player $\forall$ wins the fixpoint game.

**Observation 3.1.1.** *In the previous example, $\exists$'s positions are not represented by pairs $(b, i)$ because the game is played on a single equation, and thus the index would be useless. For the same reason, the positions of the universal player are simple non-vector items.*

## 3.2 Encoding Strategies as *Progress Measures*

In this section we summarise and account for the notion of *Progress Measure*, which [2] defined (and adjusted for the Fixpoint Game) along the lines of [13], a seminal work we extensively drew from in our brief recapitulation about Parity Games as well.

**Definition 3.2.1.** *(**Progress Measure**).*
*Let* L *be a continuous lattice and let* E *be a system of equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. *A* Progress Measure *for* E *is a function* $R\colon B_L \to \underline{m} \to \mathbb{N}^m \cup \top$, *such that for all* $b \in B_L$, $i \in \underline{m}$, *either* $R(b)(i) = \top$ *or there exists* $\boldsymbol{l} \in \mathbf{E}(b, i)$ *such that for all* $(b', j) \in \mathbf{A}(\boldsymbol{l})$ *it holds:*

- *if* $\eta_i = \mu$ *then* $R(b)(i) \succ_i R(b')(j)$;
- *if* $\eta_i = \nu$ *then* $R(b)(i) \succeq_i R(b')(j)$.

A Progress Measure maps any pair $(b, i)$ (with $b \in B_L$ and $i \in \underline{m}$) to an $m$-tuple of natural numbers, where $m$ is the number of equations in $E$. The formal definition given above could seem somewhat unclear at first glance, but its meaning can be grasped as follows: intuitively, whenever $R(b)(i) \neq \top$, the Progress Measure $R$ provides an *evidence of the existence of a winning strategy* for $\exists$ in a play starting from $(b, i)$.
The following lemma (whose proof can be found in [2]) justifies the need for Progress Measures in parity games by highlighting their deeper game-theoretic meaning.

**Lemma 3.2.1.** *(**Progress Measures represent Strategies**).*
*Let* L *be a continuous lattice and let* E *be a system of equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *with solution* $\boldsymbol{u}$. *For any* $b \in B_L$ *and* $i \in \underline{m}$, *if there exists some natural number* $n$ *and a Progress Measure* $R$ *such that* $R(b)(i) \preceq_i (n, ..., n)$, *then* $b \sqsubseteq u_i$.

The point of the above lemma is twofold: first, it says that Progress Measures provide *sound characterisations of the solution*, but it should be noted that *they are not complete*, since whenever $R(b)(i) = \top$, no information can be inferred on $(b, i)$ and thus on the solution of the system. Stronger conditions on the definition are therefore needed in order to achieve completeness.

**Definition 3.2.2.** *(**Complete Progress Measures**).*
*Let* L *be a continuous lattice and let* E *be a system of equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *with solution* $\boldsymbol{u}$. *A Progress Measure* $R\colon B_L \to \underline{m} \to [n]^m \cup \top$ *(where* $n \in \mathbb{N}$*) is called* complete *if for all* $b \in B_L$ *and* $i \in \underline{m}$, *if* $b \sqsubseteq u_i$ *then* $R(b)(i) \preceq_i (n, ..., n)$.

### 3.2.1 A Fixpoint Characterisation

Definition 3.2.1 naturally induces a characterisation of Progress Measures as the least solution of a system of equations over tuples of naturals. A key observation is that whenever a complete Progress Measure exists, its components are bounded by the height of the lattice.

**Definition 3.2.3.** *(**Progress Measure Equations**).*
*Let* L *be a continuous lattice and let* E *be a system of equations over* L *of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$. *Let* $\boldsymbol{\delta_i^\eta}$, *with* $i \in \underline{m}$, *be, for* $\eta = \nu$, *the null vector and, for* $\eta = \mu$, *the vector such that:*

* $\delta_j = 0$ *if* $j \neq i$;
* $\delta_j = 1$ *if* $j = i$.

*The* Progress Measure Equations *for* E *over the lattice* $[\lambda_{\mathrm{L}}]^m \cup \top$, *are defined, for* $b \in B_{\mathrm{L}}$, $i \in \underline{m}$, *as:*

* $R(b)(i) = min_{\preceq_i}\{sup\{R(b')(j) + \boldsymbol{\delta_i^{\eta_i}} \mid (b', j) \in \mathbf{A}(\boldsymbol{l})\} \mid \boldsymbol{l} \in \mathbf{E}(b, i)\}$

$\Phi_E$ *will denote the corresponding endofunction on* $\mathrm{L} \to \underline{m} \to [\lambda_L]^m \cup \top$ *which is defined by:*

* $\Phi_E(R)(b)(i) = min_{\preceq_i}\{sup\{R(b')(j) + \boldsymbol{\delta_i^{\eta_i}} \mid (b', j) \in \mathbf{A}(\boldsymbol{l})\} \mid \boldsymbol{l} \in \mathbf{E}(b, i)\}$

**Theorem 3.2.1.** *The last definition, along with the fact that the set of Progress Measures forms a lattice allows us to obtain a complete Progress Measure as a (least) fixpoint of the functional* $\Phi_E$.

### 3.2.2 Efficiency issues

As regards the theory we needed, we could say that the results presented in the previous section are enough and adequate, since they reduce the problem to the solution of a system of least fixpoint equations. Progress Measures would in turn allow one to prove properties of the solution of the equational system, which is exactly what we sought. However, the definitions such results were based on would lead to much inefficiency in the computation, which is due to the huge amount of possible moves that can be made by the existential player: in fact, Progress Measure equations were defined in such a way that they need a lookup to the whole set of moves. This hindrance can be easily overcome by exploiting the following observation (preceded by a useful definition).

**Definition 3.2.4.** *(**Upward-closed set**).*
*An* upward-closed set *of a* partially ordered set $(X, \sqsubseteq)$ *is a subset* $U$ *with the property that, if* $x \in U$ *and* $x \sqsubseteq y$, *then* $y \in U$ *(the notion of* downward-closed set *is dual).*

**Observation 3.2.1.** *(**The set of moves is much smaller**).*
*Given a system* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *on a lattice* L, *from a position* $(b, i)$, *given any move* $\boldsymbol{l} \in \mathbf{E}(b, i)$ *for player* $\exists$, *i.e. any tuple such that* $b \sqsubseteq f_i(\boldsymbol{l})$, *it is clear that any* $\boldsymbol{l'}$ *such that* $\boldsymbol{l} \sqsubseteq \boldsymbol{l'}$ *is a valid move for* $\exists$, *since the monotonicity of* $f_i$ *implies that* $b \sqsubseteq f_i(\boldsymbol{l}) \sqsubseteq f_i(\boldsymbol{l'})$, *that is to say,* $\mathbf{E}(b, i)$ *is an* upward-closed *set. Since player* $\exists$ *has to descend as much as possible in order to win, playing larger elements would be inconvenient from its point of view.*

The calculation can be made faster in an automatic way by introducing the notion of *selection*, which we restrict to the case of a single function in the beginning.

**Definition 3.2.5.** *(**Selection for a function f**).*
*Let* L *be a lattice. Given a monotone function* $f : \mathrm{L}^m \to \mathrm{L}$*, a* selection *for* $f$ *is a function* $\sigma : B_{\mathrm{L}} \to 2^{\mathrm{L}^m}$ *such that for all* $b \in B_{\mathrm{L}}$ *the following equality holds:*

$$* \ \mathbf{E}(b, f) = \{\boldsymbol{l} \mid \boldsymbol{l} \in \mathrm{L}^m \wedge b \sqsubseteq f(\boldsymbol{l})\} = \uparrow \sigma(b)$$

*Given a system* E *of* $m$ *equations of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$*, a* selection *for* E *is an* $m$*-tuple of functions* $\boldsymbol{\sigma}$ *such that, for each* $i \in \underline{m}$*, the function* $\sigma_i$ *is a selection for* $f_i$*.*

Intuitively, given an element $b$ and a function $f$, the *selection* of $b$ ($\sigma(b)$) will provide the "smallest" way (that is, a list of assigments to $f$'s parameters) to include (or *cover*) $b$ by means of $f$. Since the concept of selection will be essential in the development of this work, below we propose an example where the functions $f$ are $\mu$-calculus' basic operators, so as to make the notion clearer and closer to a practical application.

**Example 3.2.1.** *(**Selections: an application**).*
*Here we show the selections for* $\mu$*-calculus' basic operators, which are just four:* $\wedge, \vee, \Diamond$ *and* $\Box$*. The generic operators* $\wedge$ *and* $\vee$ *become the usual set operations* $\cup$ *and* $\cap$ *when the formula is on a real transition system. Thus, given a transition system* $(\mathbb{S}, \to)$*, consider the powerset lattice* $L = 2^{\mathbb{S}}$ *ordered according to* $\subseteq$*, with basis* $B_L = \{\{s\} \mid s \in \mathbb{S}\}$*. The least selection for each operator is given below.*

1. *Let* $f$ *be the* $\cup$ *operator, that is to say,* $f$*:* $L^2 \to L$ *such that* $f(x_1, x_2) = x_1 \cup x_2$*. Then the least selection* $\sigma : B_L \to 2^{L^2}$ *for* $f$ *is* $\sigma(\{s\}) = \{(\emptyset, \{s\}), (\{s\}, \emptyset)\}$*. This value for* $\sigma$ *means that there are two ways to cover* $\{s\}$ *in the smallest manner: the first assigns the empty set to* $x_1$ *and the set* $\{s\}$ *to* $x_2$*, whereas the latter indicates the opposite assigment. In fact, if we try to compute* $f$ *on these assigments we get:* $\emptyset \cup \{s\} = \{s\} \cup \emptyset = \{s\}$*, which is obviously the smallest set that includes* $\{s\}$*.*

2. *Let* $f$ *be the* $\cap$ *operator, that is to say,* $f$*:* $L^2 \to L$ *such that* $f(x_1, x_2) = x_1 \cap x_2$*. Then the least selection* $\sigma : B_L \to 2^{L^2}$ *for* $f$ *is* $\sigma(\{s\}) = \{(\{s\}, \{s\})\}$*. This value for* $\sigma$ *means that there is just one single way to cover* $\{s\}$ *in the smallest manner: the value* $\{s\}$ *is assigned to both* $x_1$ *and* $x_2$*. In fact, if we try to compute* $f$ *on this assigment we get:* $\{s\} \cap \{s\} = \{s\}$*, which is obviously the smallest set that includes* $\{s\}$*.*

3. *Let* $f$ *be the* $\Diamond$ *operator, that is to say,* $f$*:* $L \to L$ *such that* $f(x) = \Diamond x$*. Then the least selection* $\sigma : B_L \to 2^L$ *for* $f$ *is* $\sigma(\{s\}) = \{\{s'\} \mid s \to s'\}$*. The meaning of this selection can be realised better by looking at the operator's semantics, which is the following:* $\Diamond x = \{s \mid \exists s \to s' \wedge s' \in x\}$*. The above selection tells us that we can assign to* $x$ *any state* $s'$ *(actually* $\{s'\}$*)* $s$ *is able to reach. In fact, if we try to compute* $f$ *on one of such* $\{s'\}$ *we get:*
   $\Diamond\{s'\} = \{s\} \cup \{$*any other state able to reach* $s'\}$*, which is clearly the smallest way to include* $\{s\}$*.*

4. *Let* $f$ *be the* $\Box$ *operator, that is to say,* $f$*:* $L \to L$ *such that* $f(x) = \Box x$*. Then the least selection* $\sigma : B_L \to 2^L$ *for* $f$ *is* $\sigma(\{s\}) = \{\{s' \mid s \to s'\}\}$*. As in the previous*

*case, let us look at the operator's semantics:* $\Box x = \{s \mid \forall s \to s', s' \in x\}$. *The above selection tells us that we must assign to* $x$ *the set of states* $s$ *is able to reach. In fact, if we try to compute* $f$ *on such set we get:*
$\Box\{\{s' \mid s \to s'\}\} = \{s\} \cup \{$*any other state always reaching* $s'\}$, *which is clearly the smallest way to include* $\{s\}$.

It should be noted that only the complete knowledge of $\mu$-calculus operators' semantics on a transition system permitted us to work out their selections.
The next lemma gives sufficient conditions for a function to admit a least selection. It is always true when the lattice is finite.

**Lemma 3.2.2.** *(**Existence of Least Selections**).*
*Let* L *be a finite lattice with a basis* $B_{\mathrm{L}}$ *and let f:* $\mathrm{L}^m \to \mathrm{L}$ *be a monotone function. If f preserves the infimum of descending chains, then it admits a least selection* $\sigma_m$ *that maps each* $b \in B_{\mathrm{L}}$ *to the set of minimal elements of* $\mathbf{E}(b, f)$.

**Observation 3.2.2.** *(**Winning Strategies and Selections**).*
*If a winning strategy exists for the fixpoint game, we can find one such strategy that is a subset of any given selection.*

**Observation 3.2.3.** *(**Selection Composition**).*
*If a function f consists of the composition of some component functions, then a selection for f can be derived from the components' selections.*

#### 3.2.2.1 A framework for expressing *E(b,i)* in logical form

We have observed that the set of possible moves of the existential player constitutes an upward-closed set. It is known that the extensive notation of (minimal) selections has a size which is exponential in the arity of $f$ (the function on which the selection is computed). This exponential explosion is explained better by means of an example.

**Example 3.2.2.** *(**The size of selections grows exponentially**).*

- *consider the function* $f(x_1, x_2, x_3, x_4) = (x_1 \cup x_2) \cap (x_3 \cup x_4)$, *on our usual lattice. Its minimal selection is:*
  $\sigma(\{s\}) = \{(\{s\}, \emptyset, \{s\}, \emptyset), (\{s\}, \emptyset, \emptyset, \{s\}), (\emptyset, \{s\}, \emptyset, \{s\}), (\emptyset, \{s\}, \{s\}, \emptyset)\}$.

- *now consider a larger arity:*
  $f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 \cup x_2) \cap (x_3 \cup x_4) \cap (x_5 \cup x_6)$.
  *Its minimal selection is:*
  $\sigma(\{s\}) =$
  $\{(\{s\}, \emptyset, \{s\}, \emptyset, \{s\}, \emptyset), (\{s\}, \emptyset, \{s\}, \emptyset, \emptyset, \{s\}),$
  $(\{s\}, \emptyset, \emptyset, \{s\}, \{s\}, \emptyset), (\{s\}, \emptyset, \emptyset, \{s\}, \emptyset, \{s\}),$
  $(\emptyset, \{s\}, \{s\}, \emptyset, \{s\}, \emptyset), (\emptyset, \{s\}, \{s\}, \emptyset, \emptyset, \{s\}),$
  $(\emptyset, \{s\}, \emptyset, \{s\}, \{s\}, \emptyset), (\emptyset, \{s\}, \emptyset, \{s\}, \emptyset, \{s\})\}$.
  *As you can see, the last selection contains* 8 *tuples. Note that* $8 = 2^{6/2}$, *where* 6 *is the arity of f. The combinatorial explosion caused by the extensive writing of selections can be easily realised thanks to this example.*

**Definition 3.2.6.** *(**The Logical Framework**).*
*Let* L *be a lattice and let* $B_{\mathrm{L}}$ *be a basis for* L*. Given* $m \in \mathbb{N}$*, the* logic $\mathcal{L}_m(B_{\mathrm{L}})$ *has the following inductive definition, where* $b \in B_{\mathrm{L}}$ *and* $j \in \underline{m}$*:*

$$\phi ::= [b, j] \mid \bigvee_{k \in K} \phi_k \mid \bigwedge_{k \in K} \phi_k$$

The connection between this language and upward-closed sets is attained by a compositional semantics in the way explained below.

**Definition 3.2.7.** *(**A Semantics for the Logical Framework**).*
*The semantics of a formula* $\phi$ *is an upward-closed set* $[\![\phi]\!] \subseteq L^m$ *(the* $[\![]\!]$ *operator returns the semantic meaning of its argument), defined according to the structure of the formula as follows:*

* $[\![[b, j]]\!] = \{\boldsymbol{l} \in L^m \mid b \sqsubseteq l_j\}$;
* $[\![\bigvee_{k \in K} \phi_k]\!] = \bigcup_{k \in K} [\![\phi_k]\!]$;
* $[\![\bigwedge_{k \in K} \phi_k]\!] = \bigcap_{k \in K} [\![\phi_k]\!]$.

Intuitively, the semantic meaning of an atom $[b, j]$ is an upward-closed set, and therefore the whole semantics (which strictly follows the language's syntax) yields upward-closed sets.
As regards the logic's expressive power, the next lemma states that each upward-closed set can be denoted by way of a formula of the language in question.

**Lemma 3.2.3.** *(**Formulae for Upward-Closed Sets**).*
*Let* L *be a lattice with basis* $B_{\mathrm{L}}$ *and let* $X \subseteq \mathrm{L}^m$ *be* upward-closed*. Then* $X = [\![\phi]\!]$*, where* $\phi$ *is the formula in* $\mathcal{L}_m(B_{\mathrm{L}})$ *defined as follows:*

$$\phi = \bigvee_{l \in X} \bigwedge \{[b, j] \mid j \in \underline{m} \wedge b \sqsubseteq l_j\}$$

It should be observed that such formulae can even consist of infinite compositions of boolean operators; however, we will focus on finite formulae since our work does not consider any lattices other than finite ones, where formulae are certainly finite. The logical framework we presented so far can be profitably exploited for writing succinctly the set $\mathbf{E}(b, f)$ and thus the set of moves that can be done by the existential player ($\mathbf{E}(b, i)$). We remind the reader that the main objective of the logic is expressing selections in a compact way, thus enabling a more efficient solution of Progress Measure equations, which depend on $\mathbf{E}(b, i)$ by definition. Solving Progress Measure equations efficiently requires an alternative definition of such equations, which is based on the fundamental concept defined in the following definition.

**Definition 3.2.8.** *(**Symbolic** $\exists$**-moves**).*
*Let* L *be a lattice and let* $f : \mathrm{L}^m \to \mathrm{L}$ *be a monotone function. A* symbolic $\exists$-move *for* $f$ *is a family* $(\phi_b)_{b\in B_\mathrm{L}}$ *of formulae in* $\mathcal{L}_m(B_\mathrm{L})$ *such that* $[\![\phi_b]\!] = \mathbf{E}(b, f)$ *for all* $b \in B_\mathrm{L}$.
*The extension to systems is natural: if* E *is a system of* $\underline{m}$ *equations of the kind* $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$ *over a lattice* L, *a* symbolic $\exists$-move *for* E *is a family of formulae* $(\phi_b)^i_{b\in B_\mathrm{L}, i\in\underline{m}}$ *such that for all* $i \in \underline{m}$, *the family* $(\phi_b)_{b\in B_\mathrm{L}}$ *is a* symbolic $\exists$-move *for* $f_i$.

A noteworthy fact: symbolic $\exists$-moves can be composed to generate more complex symbolic $\exists$-moves, that is to say, given a function $g$ arising from the composition of functions $f_1, ..., f_n$, the symbolic $\exists$-move for $g$ can be obtained as the composition of the symbolic $\exists$-moves for $f_1, ..., f_n$. The following lemma formalises this intuition.

**Lemma 3.2.4.** *(**A Procedure for Composing Symbolic** $\exists$**-moves**).*
*Let* L *be a lattice with a basis* $B_\mathrm{L}$, *let* $f : \mathrm{L}^n \to \mathrm{L}$, $f_j : \mathrm{L}^m \to \mathrm{L}$ *for* $j \in \underline{n}$ *be monotone functions and let* $(\phi_b)_{b\in B_\mathrm{L}}$, $(\phi_b)^j_{b\in B_\mathrm{L}, j\in\underline{n}}$ *be symbolic* $\exists$*-moves for* $f, f_1, ..., f_n$. *Consider the function* $h : \mathrm{L}^m \to \mathrm{L}$ *obtained as the composition* $h(\boldsymbol{l}) = f(f_1((\boldsymbol{l}), ..., f_n(\boldsymbol{l})))$.
*Define* $(\phi'_b)_{b\in B_\mathrm{L}}$ *as follows: for each* $b \in B_\mathrm{L}$, *the formula* $\phi'_b$ *is obtained from* $\phi_b$ *by replacing each occurrence of* $[b', j]$ *(each atom) by* $\phi^j_{b'}$. *Then* $(\phi'_b)_{b\in B_L}$ *is a symbolic* $\exists$*-move for* $h$.

The next example will apply the procedure presented above on a simple function arising from the composition of boolean operators.

**Example 3.2.3.** *Consider the boolean lattice* $L = \{false, true\}$ *and the function* $f : L^3 \to L$ *defined as:* $f(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$. *The function* $f$ *clearly stems from the composition of two simpler functions (namely, the two boolean operators),* $f_1 = \wedge$ *and* $f_2 = \vee$*; therefore, for the sake of clarity,* $f$ *can be written as* $f_1(x_1, f_2(x_2, x_3))$. *Furthermore, each variable* $x_i$ *should be replaced by the corresponding projection operator* $\pi_i$ *in order to apply the procedure illustrated above: as a consequence, we refine the expression for* $f$ *as* $f_1(\pi_1(\vec{x}), f_2(\pi_2(\vec{x}), \pi_3(\vec{x})))$.
*The symbolic* $\exists$*-moves for* $f_1$, $f_2$ *and for projection operators are listed below (note that, for this kind of operators, symbolic* $\exists$*-moves do not depend on the actual basis element, and therefore we will use the parameter* $b$ *as general basis element):*

1. $\phi^{f_1}_b = [b, 1] \wedge [b, 2]$;
2. $\phi^{f_2}_b = [b, 1] \vee [b, 2]$;
3. $\phi^{\pi_i}_b = [b, i]$.

*Now, we can apply the procedure on* $\phi^{f_1}_b$, $\phi^{f_2}_b$ *and the three operators* $\pi_i$ *to obtain the final composite symbolic* $\exists$*-move, which is* $\phi^f_b = [b, 1] \wedge ([b, 2] \vee [b, 3])$. *In this instance, the final formula resembles very much the function's structure, but this fact is not true in general.*

The structure of a symbolic $\exists$-move indicates which computation should be carried out in order to evaluate the fixpoint equations for Progress Measures. The intuitive recipe is based on the formula's structure, this way:

* replace every disjunction with a minimum;
* replace every conjunction with a supremum;
* an atom translates to a straightforward lookup to the corresponding Progress Measure.

**Theorem 3.2.2.** *(**Progress Measures in terms of Symbolic $\exists$-Moves**).*
*Let* E *be a system of* $m$ *equations over a lattice* L *and let* $B_{\mathrm{L}}$ *be a basis for* L*. Let* $(\phi_b)^i_{b \in B_{\mathrm{L}}, i \in \underline{m}}$ *be a symbolic* $\exists$*-move for* E*.*
*Then,* $\forall b \in B_{\mathrm{L}}, \forall i \in \underline{m}$*, the system of fixpoint Progress Measure equations can be written as:*

* $$R(b)(i) = R^i_{\phi^i_b},$$

*where* $R^i_{\phi^i_b}$ *has the following inductive (on the structure of* $\phi^i_b$*) definition:*

* $$R^i_{[b,j]} = min_{\preceq_i}\{R(b)(j) + \boldsymbol{\delta}^{\boldsymbol{\eta}_i}_{\boldsymbol{i}}\};$$
* $$R^i_{\bigvee_{k \in K} \phi_k} = min_{k \in K} R^i_{\phi_k};$$
* $$R^i_{\bigwedge_{k \in K} \phi_k} = sup_{k \in K} R^i_{\phi_k}.$$

Some observations should be made: the first equation holds if the element $b$ is compact, but this fact is true whenever the lattice is finite, which is the case we are studying. As regards the $min_{\preceq_i}$ operator in the first equation, it does not really range over a set of items: in fact, it is semantically equivalent to the function that, given a vector, sets to zero all its elements with an index less than $i$.

An accurate complexity analysis of computing Progress Measures in terms of symbolic $\exists$-moves will constitute a large part of the next chapter, where we will also describe the fundamental graph-theoretic and fixpoint algorithms on which our tool will rely.

CHAPTER 4

# Algorithms

In the previous chapter, the notion of Progress Measure and its utility for fixpoint games have been presented exhaustively. Progress Measures were first used for devising an algorithm for solving parity games: as a result, it was found that the complexity of model-checking the $\mu$-calculus is polynomial in the number of states and exponential in half of the alternation depth of a formula. However, up until now, a complexity analysis for our setting, which is based on selections and symbolic $\exists$-moves, has not yet been introduced.
With respect to possible optimisations, we will see that detecting dependencies among logical formulae will be paramount in making the fixpoint computation faster.

## 4.1 Dependency Graphs

As we saw in the last theorem of the previous chapter, the shape of the logical formulae in symbolic $\exists$-moves dictates the way the values of $R$ at various game positions $(b, i)$ are related among themselves.
The intuition on dependency among formulae can be formalised in a graph-theoretic fashion, as the following definition suggests.

**Definition 4.1.1.** *(**Dependency Graph** [2]).*
*Given two game positions, $(b, i), (b', j) \in B_L \times \underline{m}$ of the existential player $\exists$, we say that $(b, i)$ is a* predecessor *of $(b', j)$ if $[b', j]$ occurs in $\phi_b^i$.*
*We will write $pred(b', j)$ for the set of predecessors of $(b', j)$. If $(b, i)$ is a* predecessor *of $(b', j)$ then we call the pair $((b, i), (b', j))$ an* edge*, and the graph consisting of game positions as nodes and the mentioned pairs as edges is referred to as* Dependency Graph *for the system* E*.*
*We could say that in this kind of graphs, a node depends on its successors.*

First and foremost, a bound on the number of edges is given. Its utility will be clear in the subsequent complexity analysis.

**Lemma 4.1.1.** *(**An upper bound on the number of edges**).*
*The number $e$ of edges in the Dependency Graph for a system* E *is such that $e \leq min\{|B_L| \cdot m \cdot s, (|B_L| \cdot m)^2\}$, where $m$ is the number of equations and $s$ is the bound on the size of symbolic $\exists$-moves.*

***Proof***:
Game positions are at most $|B_L| \cdot m$, hence the number of edges is at most equal to $(B_L \cdot m)^2$. In addition, for each game position $(b, i)$, the number of outgoing edges is bounded by the size of the symbolic $\exists$-move $\phi_b^i$. Thus, $e$ does not exceed the least of the two numbers, which is the thesis. □

As with any fixpoint computation, Progress Measure equations can be solved using a suitable instantiation of the *Worklist Algorithm* (see [19] for a thorough walkthrough), which is known for ensuring better efficiency than other fixpoint algorithms, because it takes dependencies among variables (which in our case correspond to dependencies among game positions) into account, thus limiting the number of redundant function applications. However, its better efficiency comes at the cost of a painstaking initialisation and organisation of the *worklist*, which by itself constitutes an optimisation ([6]) we will discuss further in the next sections.

**Theorem 4.1.1.** ***(The Time Complexity of Computing Progress Measures)***.
*The time complexity for finding the least fixpoint of Progress Measure equations for a system* E *is* $O(s \cdot k \cdot e \cdot h)$*, where:*

- *s is the bound on the size of symbolic $\exists$-moves;*
- *k is the number of least fixpoint equations ($\mu$-equations) of* E*;*
- *e is the number of edges in the Dependency Graph;*
- *h is equal to* $(\lambda_L + 1)^k + 1$.

### 4.1.1 Procedures

A couple of procedures for producing a dependency graph are presented here in pseudocode. We opted to represent graphs by means of adjacency lists, since the precedence relation described in Definition 4.1.1 can be easily interpreted as an adjacency relation between pair of vertices.

**Data:** a formula $\phi_b^i$, a *pred* list for each game position $(b, i)$
**Result:** current *pred* lists for the pairs $(b, i)$ present in the logical clauses of the current formula
**for** *each* $[b', j]$ *in* $\phi_b^i$ **do**
 | pred($b'$,$j$) := pred($b'$,$j$) ∪ ($b$,$i$);
**end**

**Algorithm 1:** The *addToPred* routine.

The following algorithm calls the one above to fill all the *pred* lists, so as to yield a complete encoding of the final dependency graph for a given symbolic $\exists$-move.

**Data:** a symbolic ∃-move for a system, an empty *pred* list for each game position $(b, i)$
**Result:** full *pred* list for each position $(b, i)$
**for** *i = 1; i <= m* **do**
  **for** *each b in* $B_L$ **do**
    addToPred($\phi_b^i$, list of pred lists);
  **end**
**end**

**Algorithm 2:** The *generateDepGraph* routine.

Generating a dependency graph with the previous two routines requires only a single read of the set of symbolic ∃-moves.

**Example 4.1.1.** *(**Running** **generateDepGraph** **on a set of moves**).* *Consider the following list of symbolic ∃-moves:*

* $\phi^1_{\{a\}} = [\{a\}, 1] \vee [\{b\}, 1]$;
* $\phi^1_{\{b\}} = true$;
* $\phi^2_{\{a\}} = [\{a\}, 1] \wedge [\{a\}, 2] \wedge [\{b, 2\}]$;
* $\phi^2_{\{b\}} = [\{b\}, 1] \wedge [\{b\}, 2]$.

*Running the procedure called* generateDepGraph *on such set of moves generates the dependency graph depicted in the figure below. Observe that node* $(\{b\}, 1)$*'s successors have been omitted since this set of nodes includes all the nodes of the graph (because* $\phi^1_{\{b\}} = true$*, where true represents all possible atoms).*

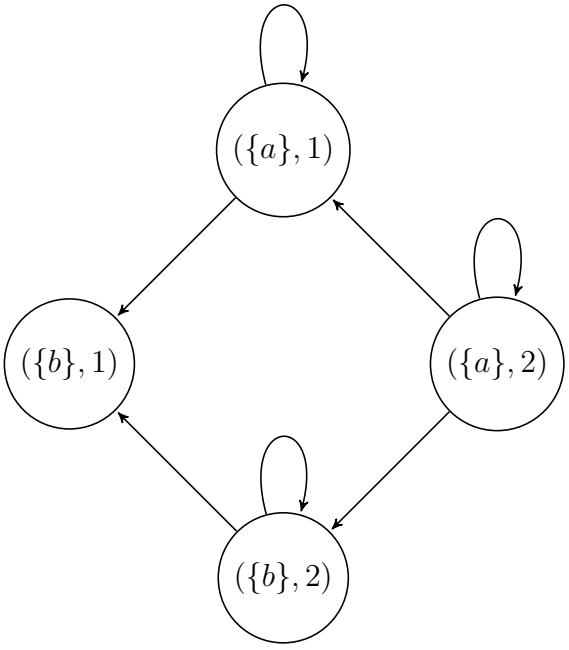



**Figure 4.1:** A dependency graph.

## 4.2 Computing the Fixpoint

We aim at solving systems of Progress Measure equations using a suitable fixpoint algorithm. Obviously, a system of equations can be seen as a function on multiple components. It is known that a naive implementation of the Knaster-Tarski algorithm for computing least fixpoints leads to inefficiency due to unneeded function applications, when the function is defined on multiple components. We therefore leveraged the *Worklist Algorithm* to work out fixpoints, relying on the dependency graph computed earlier. As the name suggests, this algorithm keeps a list of items which depend on modified items, with the purpose of limiting the number of function applications as far as possible. One of the items (often the first one) of this *worklist* is extracted at each iteration, and the loop halts as soon as the list becomes empty.
An example of the kind of systems we would like to solve by means of a fixpoint algorithm is the one reported below:

$$\begin{cases} R(\{a\},1) =_\mu R^1_{\phi^1_{\{a\}}} \\ R(\{a\},2) =_\mu R^2_{\phi^2_{\{a\}}} \\ R(\{b\},1) =_\mu R^1_{\phi^1_{\{b\}}} \\ R(\{b\},2) =_\mu R^2_{\phi^2_{\{b\}}} \\ R(\{c\},1) =_\mu R^1_{\phi^1_{\{c\}}} \\ R(\{c\},2) =_\mu R^2_{\phi^2_{\{c\}}} \end{cases}$$

where each Progress Measure is given in terms of a known symbolic $\exists$-move.

Below we report three pseudocodes, one for each way of computing the fixpoint, ranging from the basic Knaster-Tarski ([9]) procedure to *Chaotic Iteration* ([19]) and the Worklist Algorithm. Observe that the first two algorithms share a common, remarkable issue: for any component, the information is recomputed at each iteration, even if it is known in advance that the information cannot change, because it depends on unaltered information.

**Data:** $X := [\bot, ..., \bot]$, a function $F$
**Result:** the fixpoint of $F$
**do**
  $T := X$;
  $X := F(X)$
**while** $X \neq T$;

**Algorithm 3:** The naive algorithm.

**Data:** a system of Progress Measure equations (one equation for each position $(b, i)$)
**Result:** the solution of the system
$n$ := cardinality of $B_L$;
$m$ := number of equations of the original system;
**for** *each b in* $B_L$ **do**
  **for** *i := 1; i <= m* **do**
    R($b$,$i$) := [0] * $m$ times;
  **end**
**end**
**do**
  $t_{1,1}$ := R($b_1$, 1);
  ...;
  $t_{n,m}$ := R($b_n$, $m$);
  R($b_1$, 1) = $\Phi_{1,1}$(R($b_1$, 1), ..., R($b_n$, $m$));
  R($b_n$, $m$) = $\Phi_{n,m}$(R($b_1$, 1), ..., R($b_n$, $m$));
**while** $\bigvee$ ($t_{1,1} \neq$ R($b_1$, 1),...,$t_{n,m} \neq$ R($b_n$, $m$));

**Algorithm 4:** Chaotic Iteration.

**Data:** a system of Progress Measure equations (one equation for each position $(b, i)$)
**Result:** the solution of the system
$n$ := cardinality of $B_L$;
$m$ := number of equations of the original system;
**for** *each b in* $B_L$ **do**
  **for** *i := 1; i <= m* **do**
    R($b$,$i$) := [0] * $m$ times;
  **end**
**end**
$W$ := [$v_{1,1}$, $v_{1,2}$...$v_{n,m}$];
**while** *W is not empty* **do**
  $v_{j,k}$ := head($W$);
  $W$ := tail($W$);
  [$y_1$...$y_m$] := $\Phi_{j,k}$(R($b_1$,1)...R($b_n$,$m$));
  **if** [$y_1$...$y_m$] != R($b_j$, $k$) **then**
    **for** *each v in pred*($b_j$, $k$) **do**
      $W$ := cons($v$,$W$);
    **end**
    R($b_j$,$k$) := [$y_1$...$y_m$];
  **end**
**end**

**Algorithm 5:** The Worklist Algorithm.

Some observations could help the reader comprehend the last algorithm. Let R($b_j$,$k$) be the item extracted from the worklist at a certain iteration. Now, if

applying the functional $\Phi_{j,k}$ leaves R($b_j$,$k$) unchanged, then the *if* section is not executed. This means that nothing is added to the worklist, which will therefore shrink. An intuitive invariant for the loop could be: *at each iteration, store in the worklist solely the elements that depend on other modified elements.*

## 4.3 Strongly-Connected Components of Dependency Graphs

The fundamental work by [6] contains an algorithm for model checking that handles the full modal $\mu$-calculus. The fact that such algorithm is applied on the equational system obtained from a formula makes [6] particularly appealing to us, considering that systems are indeed our thesis' main topic. As said by the paper in question, working on equational representations of $\mu$-calculus formualae facilitates the saving and reuse of intermediate results.

Firstly, consider the following definition, which will be used to define the notion of *closed subsystem.*

**Definition 4.3.1.** *An equational system* E *is* closed *if all variables in a right-hand side of some equation also appear as left-hand sides in* E

Henceforth we will follow the example reported in [6] to showcase their algorithm. Consider the following $\mu$-calculus formula:

$$\phi = \nu x_1.\mu x_2.(x_1 \vee x_2 \vee \nu x_3.\mu x_4.\nu x_5.(x_3 \wedge x_4 \wedge x_5)) \quad ,$$

and its translation into normalised equational form:

$$\begin{cases} x_1 =_\nu x_2 \\ x_2 =_\mu x_1 \vee x_3 \\ x_3 =_\mu x_2 \vee x_4 \\ x_4 =_\nu x_5 \\ x_5 =_\mu x_6 \\ x_6 =_\nu x_4 \wedge x_7 \\ x_7 =_\nu x_5 \wedge x_6 \end{cases}$$

As in the case of symbolic $\exists$-moves at the beginning of this chapter, a graph representation could aid us in addressing efficiency issues related to equational systems, since graphs can inherently provide a simple view of dependencies among variables (or game positions in the case of symbolic $\exists$-moves).
An intuitive definition of dependency graph suffices for our purpose. Intuitively, the dependency graph $G_E$ of a system of equations $E$ contains a node for each variable of the system, and there is an edge from $x_i$ to $x_j$ if the meaning of $x_i$ directly influences the meaning of $x_j$: this happens for instance when an equation contains $x_j$ as its left-hand side, and has $x_i$ in its right-hand side.

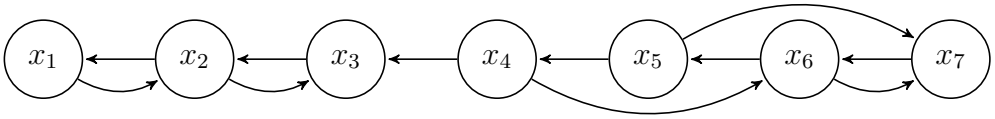



**Figure 4.2:** The dependency graph of the previous system.

Let us recall an essential definition from graph theory.

**Definition 4.3.2.** ***(Strongly-Connected Components.)***
*A directed graph is called* strongly connected *if there is a path in each direction between each pair of vertices of the graph.*
*A* strongly-connected component *of a directed graph $G$ is a subgraph that is strongly connected, and is maximal in the following sense: no additional edges or vertices from $G$ can be included in the subgraph without breaking its property of being strongly connected.*
*If each strongly-connected component is contracted to a single vertex, the resulting graph is a* directed acyclic graph*, the* condensation *of $G$.*

Some of the most popular algorithms that can be used to extract strongly-connected components from graphs are listed below:

* Kosaraju's algorithm ([7]), which uses two passes of depth-first search: the first in the original graph, the second in the transpose graph of the original one;

* Tarjan's strongly-connected components algorithm ([23]), which performs a single pass of depth-first search. We recall that the order in which the strongly-connected components are identified constitutes a reverse topological sort of the DAG formed by the strongly-connected components.

### 4.3.1 Cleaveland's procedure

1. Build the condensation graph $G_C$ of $G_E$. Observe that $G_C$ is acyclic;

2. Topologically sort $G_C$ into $G_m, ..., G_1$. The order is inferred this way: if there is an edge from $G_i$ to $G_j$, then $i > j$;

3. For each connected component, generate a closed subsystem $C_i$ containing the equations from $E$ whose left-hand sides are in $G_i$. Such equations are modified by replacing each occurrence of $X_j$ that is not a left-hand side in $G_i$ by a new atomic proposition $A_j$: this ensures that $C_i$ is closed;

4. Beginning with $C_m$, process each $C_i$ in turn.

The above procedure induces the following model checking algorithm:

1. Given $E$, determine the closed subsystems $C_1, ..., C_m$ of $E$;
2. Then process each $C_l$ in turn, beginning with $C_m$ and ending with $C_1$;
3. $[\![C_l]\!]$ is computed and stored in the relevant bit-vector components;
4. Then the atomic predicates whose semantics depend on left-hand sides in $C_l$ have their semantics initialised;
5. The algorithm terminates after $C_1$ is completed.

CHAPTER 5

# The Tool

In this chapter we describe *PMModelChecker* (the **P**rogress **M**easure **Model Checker**), the tool we designed and developed with the purpose of providing a proof of concept for the theoretical framework and algorithms presented in the previous couple of chapters. The final product is a tool for characterising the solution of a system of fixpoint equations over any lattice, where equations use a set of customisable basic monotone operators. In particular, the tool can be instantiated to a model checker for various temporal logics, given that any formula can be converted into a system of equations.

The tool's solver component will find the solution of Progress Measure equations expressed in terms of symbolic $\exists$-moves. The final fixpoint will be computed efficiently, also by taking advantage of some graph-theoretic algorithms that can be run on the dependency graph stemming from the set of symbolic $\exists$-moves (for instance, the Tarjan's algorithm for extracting the strongly connected components of a directed graph).
We point out that the operators appearing in the equations will be customisable by the final user, a feature that will make *PMModelChecker* profitable in many contexts, even beyond the propositional $\mu$-calculus and boolean equations.

## 5.1 Design steps

Processing a system of equations containing user-defined operators and then solving it via Progress Measures gave rise to the following steps in the development of our work:

* Some preprocessing steps:
    1. accept a set of user-defined operators;
    2. generate a scanner and a parser able to deal with systems of equations based on those operators;
* The actual processing and computation:
    1. accept a system of equations;

2. accept a set of symbolic ∃-moves for basic user-defined operators;
3. generate a symbolic ∃-move for each equation, relying on the basic moves input by the user;
4. generate the dependency graph of the moves generated in the previous step;
5. compute the fixpoint of the system of Progress Measure equations, using both the moves and the dependency graph in order to speed up the computation.

Evidently, the tool can be run as a model checker for the $\mu$-calculus. In this case, it accepts also a file representing the unlabelled transition system on which the original formula is being model-checked. In order to achieve a high degree of isolation among the parts of the tool, each of the above steps has been naturally translated into a software component, thus leading to a set of smaller tools, each dealing with a specific and independent task.

## 5.2 The tool's high-level structure

As we said in the previous section, each logical step has been realised by a certain component. We therefore provided a component diagram which intuitively shows the relation among such building blocks at a very high level of abstraction. The details about each component's design, such as package and class diagrams, will be reported in the next sections.

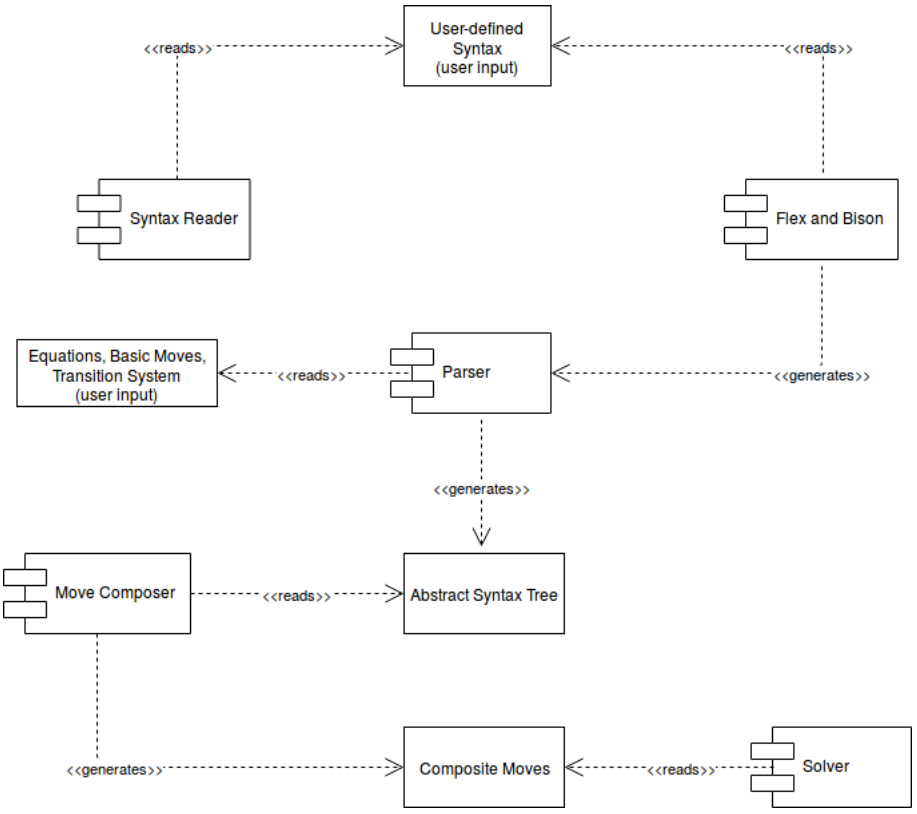



**Figure 5.1:** A high-level overview of the tool.

## 5.3 Parsing a user-defined language

The parser component (and the associated scanner for tokens) has not been written from scratch, since there are many tools available for generating parsers automatically starting from a given set of tokens and a *Context-Free Grammar* expressed in a specific language. Two such tools are Unix's *Lex* and *Yacc* (see [17] for a complete guide on these tools), to which we devoted one of the next subsections.

### 5.3.1 Customising the Parser: the *Syntax Generator*

PMModelChecker should parse and solve systems of equations of the kind $\boldsymbol{x} =_{\boldsymbol{\eta}} \boldsymbol{f}(\boldsymbol{x})$, where each $f_i$ can be obtained as a composition of basic operators $op^j_{\{k_j\}}$ (with $j \in \underline{n}$), each one with arity $k_j$.
We have already talked about the need for a customisable set of operators, a distinguishing feature that would make the tool usable in a large number of settings. However, the usual pair scanner-parser would never be able to scan and parse anything without being aware of the exact tokens and the grammar. We therefore decided to accept the set of operators along with their arities as user input, written using a simple tab-separated format, like the one in the example below (which contains a subset of $\mu$-calculus operators):

```
#a comment
#name       arity
and         2       #and operator, with arity = 2
or          2       #or operator, with arity = 2
box         1       #box operator, with arity = 1
diamond     1       #diamond operator, with arity = 1
myop        4       #myop operator, with arity = 4
```

**Table 5.1:** Configuration file for operators.

Here, for instance, the user is telling the tool that the parser component should be able to process systems whose equations could contain some of the following expressions:

* `and(expr1, expr2)`
* `or(expr1, expr2)`
* `box(expr)`
* `diamond(expr)`
* `myop(expr1, expr2, expr3, expr4)`

The first four operators we chose for the example above look very much like $\mu$-calculus' operators. The name of the operator will be used by the scanner generator

(Lex), whereas the information about the arity is needed so as to generate the Context-Free Grammar required by the automatic parser generator (Yacc).
We consequently designed the *Syntax Generator*, that is to say, a Python script whose aim is to process this kind of tab-separated input and subsequently output two files which will be employed by Lex and Yacc as their inputs. Observe that the final user will not be required to provide any tokens other than the operators, since the Python script will automatically add some regular expressions related to variable identifiers, fixpoint equalities and atoms in symbolic ∃-moves (e.g. `[x][0-9]+`, `=min`, `=max`, `[b_[0-9]+,[0-9]+`).

### 5.3.2 Lex and Yacc (a.k.a. Flex and Bison)

We often remarked on the fact that the part of the tool dealing with user input should function as a fully fledged customisable parser for systems of fixpoint equations: it could thus be stated that the tool can also be considered as a sort of *interpreter* for a programming language, or better yet, as a *calculator*.
Any compiler or interpreter for a programming language is often decomposed into two parts:

* read the source program according to a structure;
* generate the target program.

The former task (*lexing/scanning* and *parsing*) can be accomplished with major ease thanks to two famous language-design tools named *Lex* (**Lex**ical Analyzer Generator) and *Yacc* (**Y**et **A**nother **C**ompiler-**C**ompiler), which can generate large fragments of scanners and parsers in the C/C++ language starting from relatively small configuration files written in two different domain specific languages. The large amount of improvements made over the years for this couple of tools has led to newer backward compatible versions called *Flex* (a Fast Scanner Generator) and *Bison* (The YACC-compatible Parser Generator), which together constitute the actual software we relied on for our work. Anyway, for historical reasons, we will keep referring to them as Lex and Yacc.
Reading a source program according to a known structure requires the accomplishment of two independent tasks:

* split the source program into *tokens*;
* find the hierarchical tree-shaped structure of the program.

As taught by any book on basic compiler design, the former of the two tasks is called *Lexical Analysis*, whereas the latter constitutes the *Syntactic Analysis*.

#### 5.3.2.1 Lex

Lex is beneficial for writing programs whose control flow is directed by instances of regular expressions in the input stream. Any Lex input is made up of a table

of regular expressions along with corresponding *actions*, which are C/C++ code fragments that should be executed whenever a token compliant with the related regular expression is encountered during the scanning phase. Lex translates the table into a program which reads an input stream, partitioning it into strings matching with the given regular expressions. Whenever one of such strings is recognised in the input stream, the corresponding code fragment is executed. Input tokens are recognised by means of a deterministic finite automaton generated by Lex. The aforementioned C/C++ code fragments are supplied by the user and then copied verbatim into the final program generated by Lex. The diagram in the figure below sums up the steps performed by Lex to yield a working lexer. The whole

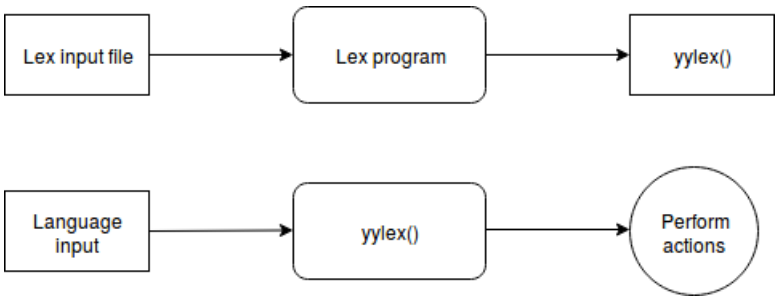



**Figure 5.2:** How Lex works.

program generated by Lex is contained in a self-standing C program in a single file, called `lex.yy.c` by default (there exists a command line option to change this denomination), which exposes the `yylex()` function.
As regards the source input accepted by Lex, its general format is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where definitions and user subroutines are often omitted.

The following listing showcases a Lex source produced by the *Syntax Generator* component after processing this list of operators. As you can see, the upper section includes a regular expression for each possible token, ranging from operators to atoms in symbolic ∃-moves, whereas the lower section contains a sequence of rules, that is to say, a mapping between tokens and small C code fragments. Essentially, such code fragments include just one assignment followed by a return instruction: actually, they constitute the link between Lex and Yacc, because such simple instructions are needed by Yacc in order to carry out its tasks and produce a working parser.

**Listing 5.1:** Lex input

```
%option interactive
%{
```

```
%}
%{
#include <string.h>
#include "expr_move.h"
#include "y.tab.h"
%}
AND         and
BOX         box
DIAMOND     diamond
MYOP        myop
OR          or
PHI         phi_[0-9]+_[a-z]+
PHI_PROP    phi_[0-9]+_[p][0-9]+
TRUE        true
FALSE       false
PROP        [p][0-9]+
EQMIN       =min
EQMAX       =max
ATOM        [b_[0-9]+,[0-9]+]
EQ          =
ID          [x][0-9]+
%{
%}
%x ERROR
%%
%{
%}
{AND}         {yylval.id = strdup(yytext); return AND_TOK;}
{BOX}         {yylval.id = strdup(yytext); return BOX_TOK;}
{PHI}         {yylval.id = strdup(yytext); return PHI_TOK;}
{PHI_PROP}    {yylval.id = strdup(yytext); return
   PHI_PROP_TOK;}
{DIAMOND}     {yylval.id = strdup(yytext); return DIAMOND_TOK
   ;}
{MYOP}        {yylval.id = strdup(yytext); return MYOP_TOK;}
{EQMIN}       {yylval.id = strdup(yytext); return EQMIN_TOK;}
{EQMAX}       {yylval.id = strdup(yytext); return EQMAX_TOK;}
{ATOM}        {yylval.id = strdup(yytext); return ATOM_TOK;}
{EQ}          {yylval.id = strdup(yytext); return EQ_TOK;}
{ID}          {yylval.id = strdup(yytext); return ID_TOK;}
{PROP}        {yylval.id = strdup(yytext); return PROP_TOK;}
{OR}          {yylval.id = strdup(yytext); return OR_TOK;}
{TRUE}        {yylval.id = strdup(yytext); return TRUE_TOK;}
{FALSE}       {yylval.id = strdup(yytext); return FALSE_TOK;}
[,()\n]       {return *yytext;}
[ \t]+        ;
. {BEGIN(ERROR);yymore();}
<ERROR>[^{DIGIT}{LETTER}+\-/*()= \t\n\f\r] { yymore();}
<ERROR>(.|\n) { yyless(yyleng-1);printf("error token: %s\n",
   yytext);
```

```
 BEGIN(INITIAL);}
%%
```

#### 5.3.2.2 A grammar for equational systems and basic symbolic ∃-moves

The structure we should impose on the input in order to let the parser process a system of equations can be expressed by a kind of formal grammar known as Context-Free Grammar: a set of *production rules* that describe all possible strings in a formal language. The grammar written in the box below describes how a user could create an input for expressing a system of boolean equations along with basic symbolic ∃-moves. A Context-Free Grammar is usually made up of both *nonterminal* (such symbols will not appear in the resulting formal language) and *terminal* symbols (such symbols do appear in the final language).
Let us list the items belonging to each set:
**Nonterminals**:

* `Sys`: it stands for the whole system of equations;
* `EqList`: it stands for a list of equations/moves;
* `Eq`: it represents an equation/move;
* `ExpEq`: it stands for the right-hand side of an equation;
* `ExpMove`: it represents the right-hand side of a move.

**Terminals**:

* the newline character;
* $\epsilon$: the empty string;
* `ID`: it stands for the regular expression `[x][0-9]+` (chosen by the scanner);
* `ATOM`: it stands for the regular expression `[b_[0-9]+,[0-9]+]` (chosen by the scanner);
* `=`$\mu$: it stands for the least fixpoint equality symbol, rendered by `=min` in the actual language;
* `=`$\nu$: it stands for the greatest fixpoint equality symbol, rendered by `=max` in the actual language;
* $\phi$: it stands for the regular expression `phi_[0-9]+_[a-z]+` (chosen by the scanner);
* the meaning of the other character terminals is obvious.

**Listing 5.2:** The grammar.

```
Sys → EqList
EqList → EqList '\n' | EqList Eq '\n' | ϵ
Eq → ID =μ ExpEq | ID =ν ExpEq | ϕ = ExpMove
ExpEq → ExpEq and ExpEq | ExpEq or ExpEq | ID
ExpEq → '(' ExpEq ')'
ExpMove → ExpMove and ExpMove | ExpMove or ExpMove
ExpMove → ATOM | '(' ExpMove ')'
```

A possible concrete (random) instance of this grammar for boolean systems is proposed here:

**Listing 5.3:** A real input.

```
x1 =min (x2 and x3) or x4
phi_0_and = [b_1,1] and [b_2,2]
x2 =max x3 and (x1 or x4)
x3 =max x1 and (x2 or x3)
phi_1_or = [b_1,1] or [b_2,2]
x4 =min (x1 or x2) and (x3 or x4)
```

Since referring to basis items only by means of their indices would look too clumsy, we decided to let the user write such items utilising readable identifiers, which should be specified in a configuration file. In the listing below, `{a}` corresponds to `b_0`, `{b}` to `b_1` and `{c}` to `b_2`.

**Listing 5.4:** A user-friendly version of the previous input.

```
# a comment
x1 =min (x2 and x3) or x4
phi_{a}_and = [{b},1] and [{c},2]
x2 =max x3 and (x1 or x4)
x3 =max x1 and (x2 or x3)
phi_{b}_or = [{b},1] or [{c},2]
x4 =min (x1 or x2) and (x3 or x4)
```

#### 5.3.2.3 Yacc

Yacc provides a general tool for describing the structure (i.e. grammar) of a language. The Yacc user expresses the grammar of the language she wants to be parsed in a domain specific language accepted by Yacc. Any Yacc input includes the following logical sections:

* a Context-Free Grammar to describe the input language;
* the C/C++ code to be invoked whenever a grammar rule is recognised in the input;
* some optional subroutines defined by the user.

It is known that any language parser depends strongly on a scanner able to process the input and split it into tokens. Obviously, such program could either be written from scratch by the user, or produced using of Lex: the only requirement is that the scanner must be contained in a function called `yylex()`, since the parser will invoke it with this name by default.

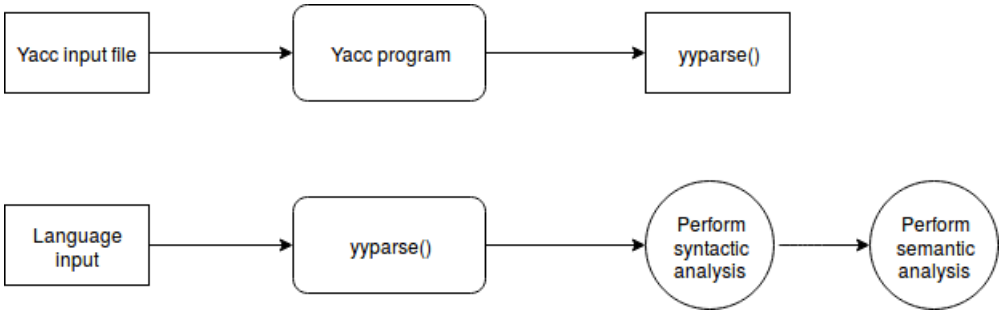



**Figure 5.3:** How Yacc works.

**Listing 5.5:** Yacc input

```
%{
#include "expr_move.h"
using namespace std;
EqSystem *root;
int yylex();
void yyerror(char *s);
%}
%start system
%union {
 char               *id;
 char               *op;
 char               *equal;
 char               *atom;
 ExpNode            *expnode;
 vector<Equation*> *equations;
 Equation           *eq;
 EqSystem           *sys;
}
%left   AND_TOK
%token  <op> AND_TOK
%left   BOX_TOK
%token  <op> BOX_TOK
%token  <id> PHI_TOK
%left   DIAMOND_TOK
%token  <op> DIAMOND_TOK
%left   FALSE_TOK
%token  <op> FALSE_TOK
%token  <id> PHI_PROP_TOK
```

```
%left   MYOP_TOK
%token  <op> MYOP_TOK
%token  <id> PROP_TOK
%token  <atom> TRUE_TOK
%token  <equal> EQMIN_TOK
%token  <equal> EQMAX_TOK
%token  <atom> ATOM_TOK
%token  <equal> EQ_TOK
%token  <id> ID_TOK
%left   OR_TOK
%token  <op> OR_TOK
%type   <expnode> expr_eq
%type   <expnode> expr_move
%type   <equations> equationlist
%type   <eq> equation
%type   <sys> system
%%
system: equationlist {$$ = new EqSystem($1); root =
    $$;}
;
equationlist: equationlist '\n'  {$$ = $1;}
  | equationlist equation '\n'  {$$ = $1;$1→
     push_back($2);}
  |  {$$ = new vector<Equation*>();}
;
equation:  ID_TOK EQMIN_TOK expr_eq  {$$ = new
   Equation($1,$2,$3);}
  | ID_TOK EQMAX_TOK expr_eq  {$$ = new Equation($1
     ,$2,$3);}
  | PHI_TOK EQ_TOK expr_move {$$ = new Equation($1,
     $2,$3);}
  | PHI_PROP_TOK EQ_TOK expr_move {$$ = new
     Equation($1,$2,$3);}
;
expr_move: expr_move AND_TOK expr_move  {$$ = new
   ExpNode($2,$1,$3);}
  | expr_move OR_TOK expr_move  {$$ = new ExpNode(
     $2, $1,$3);}
  | '(' expr_move ')'  { $$ = $2; }
  | ATOM_TOK  {$$ = new ExpNode($1,0,0);}
  | TRUE_TOK  {$$ = new ExpNode($1,0,0);}
  | FALSE_TOK {$$ = new ExpNode($1,0,0);}
;
expr_eq:  ID_TOK  {$$ = new ExpNode($1,0,0);}
  | '(' expr_eq ')'  {$$ = $2;}
  | expr_eq AND_TOK expr_eq  {$$ = new ExpNode($2,
     $1,$3);}
```

```
    | BOX_TOK expr_eq  {$$ = new ExpNode($1,$2);}
    | DIAMOND_TOK expr_eq  {$$ = new ExpNode($1,$2);}
    | MYOP_TOK expr_eq  expr_eq  expr_eq  {$$ = new
      ExpNode($1,$2,$3,$4);}
    | expr_eq OR_TOK expr_eq  {$$ = new ExpNode($2,$1
      ,$3);}
;
%%
void yyerror(char * s){printf("parse error\n");}
```

### 5.3.3 The Abstract Syntax Tree

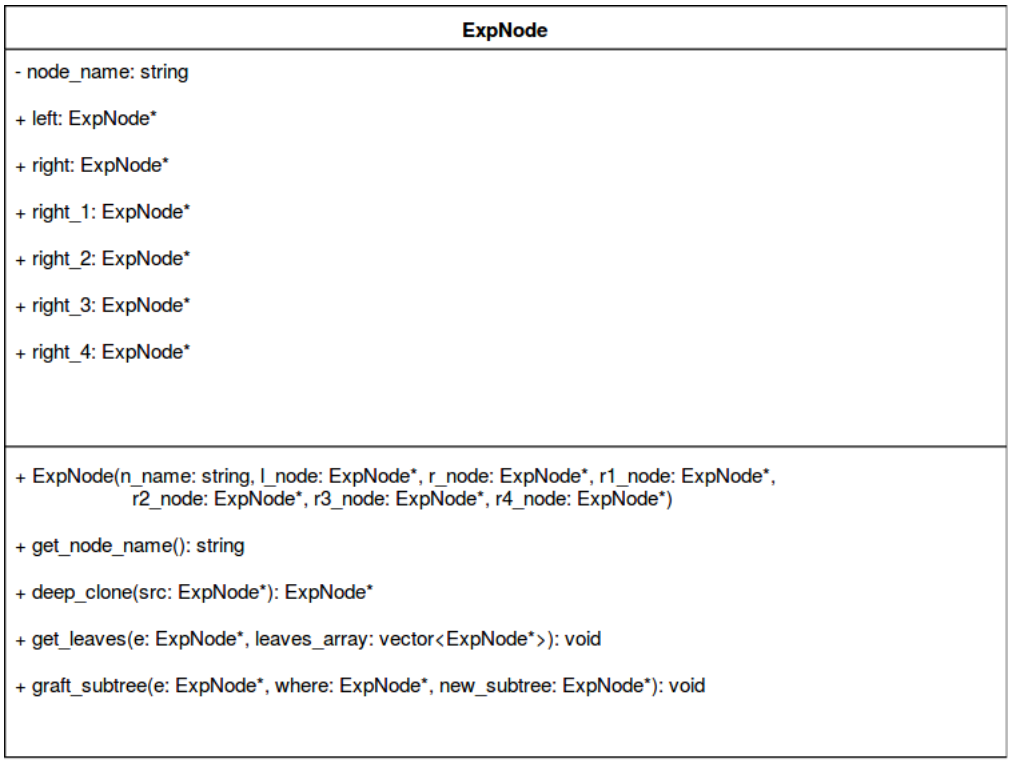
Expressions, symbolic ∃-moves, equations and systems of equations are represented by the following abstract data types (implemented as C++ classes):

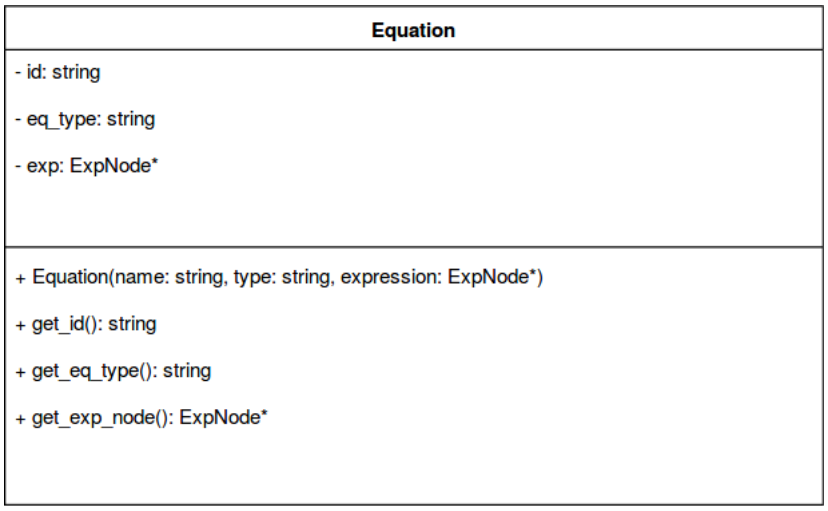
* `ExpNode`: a tree used both for common expressions of the user-defined language and for symbolic ∃-moves. Observe that `ExpNode` represents only the right-hand side of an expression/move;

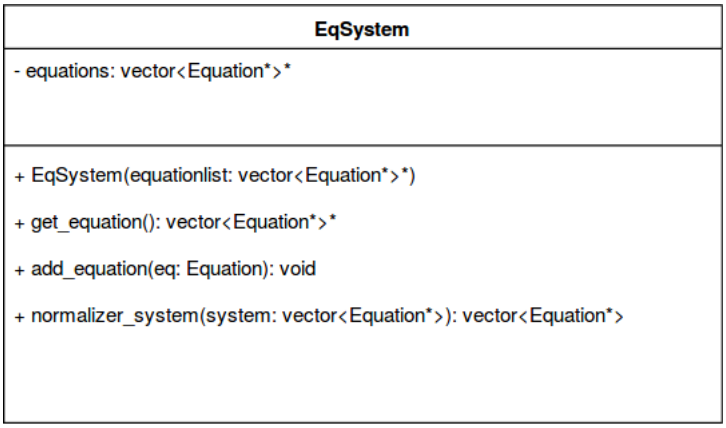
* `Equation`: adds the left-hand side (variable/move identifier and equality sign) to expressions/moves. As you already know, the available equality signs are `=`, `=min` and `=max`;

* `EqSystem`: uses a vector of pointers to `Equation` to represent a whole system of equations and symbolic ∃-moves. The *Normaliser* is implemented as a static method inside this class: given a system with equations containing compositions of operators, it transforms it into a system where each equation includes a single operator.

The Abstract Syntax Tree is produced by the Yacc-generated parser thanks to these data types.

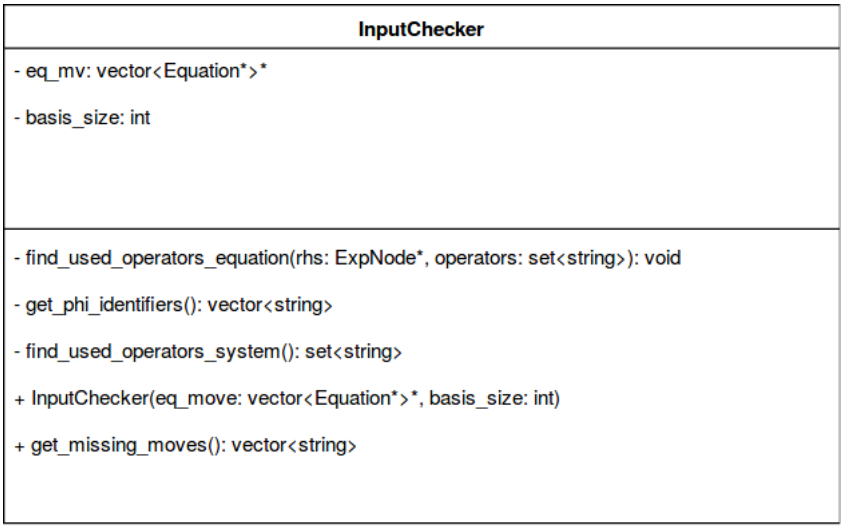

ExpNode
- node_name: string
+ left: ExpNode*
+ right: ExpNode*
+ right_1: ExpNode*
+ right_2: ExpNode*
+ right_3: ExpNode*
+ right_4: ExpNode*
+ ExpNode(n_name: string, l_node: ExpNode*, r_node: ExpNode*, r1_node: ExpNode*, r2_node: ExpNode*, r3_node: ExpNode*, r4_node: ExpNode*)
+ get_node_name(): string
+ deep_clone(src: ExpNode*): ExpNode*
+ get_leaves(e: ExpNode*, leaves_array: vector<ExpNode*>): void
+ graft_subtree(e: ExpNode*, where: ExpNode*, new_subtree: ExpNode*): void


Equation
- id: string
- eq_type: string
- exp: ExpNode*
+ Equation(name: string, type: string, expression: ExpNode*)
+ get_id(): string
+ get_eq_type(): string
+ get_exp_node(): ExpNode*


EqSystem
- equations: vector<Equation*>*
+ EqSystem(equationlist: vector<Equation*>*)
+ get_equation(): vector<Equation*>*
+ add_equation(eq: Equation): void
+ normalizer_system(system: vector<Equation*>): vector<Equation*>

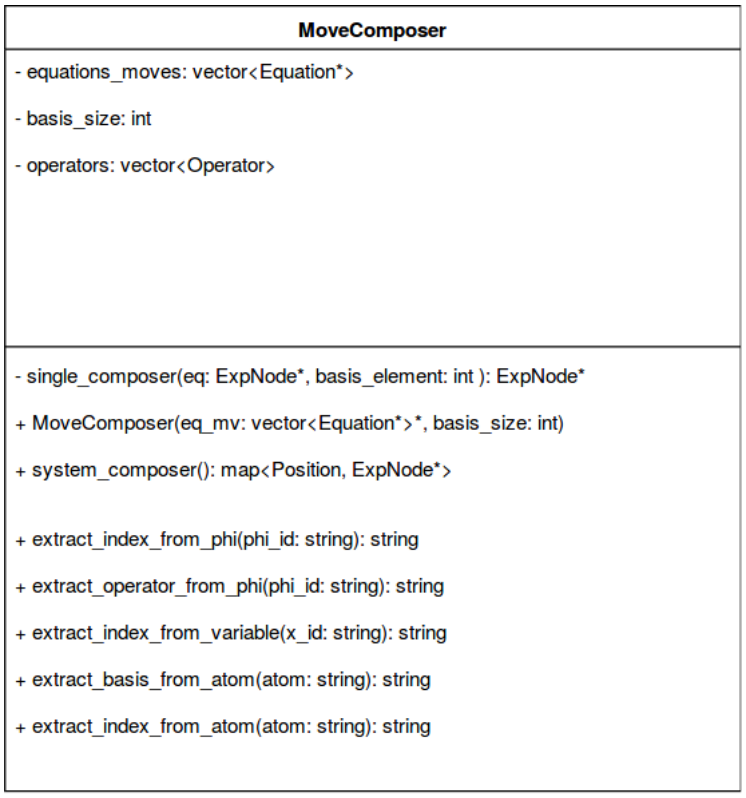
## 5.4 Finding inconsistencies in the input

Class `InputChecker` is devoted to input control. In particular, it makes sure that enough symbolic ∃-moves (one move for each basis item and for each equation index) are provided in the input file. Any inconsistency on the moves will be considered as an irreversible error and the program will exit.




## 5.5 The Move Composer

`MoveComposer` implements a recursive algorithm to build composite symbolic ∃-moves starting from simple ones. Essentially, this recursive procedure applies Lemma 3.2.4 of Chapter 3 to the binary trees (`ExpNode`) used to encode symbolic ∃-moves. Instance methods `single_composer` and `system_composer` carry out this task. The state of any `MoveComposer` instance includes also an array of the basic operators used in the input system (any operator is associated with its corresponding basic move in the same object).




## 5.6 The Solver

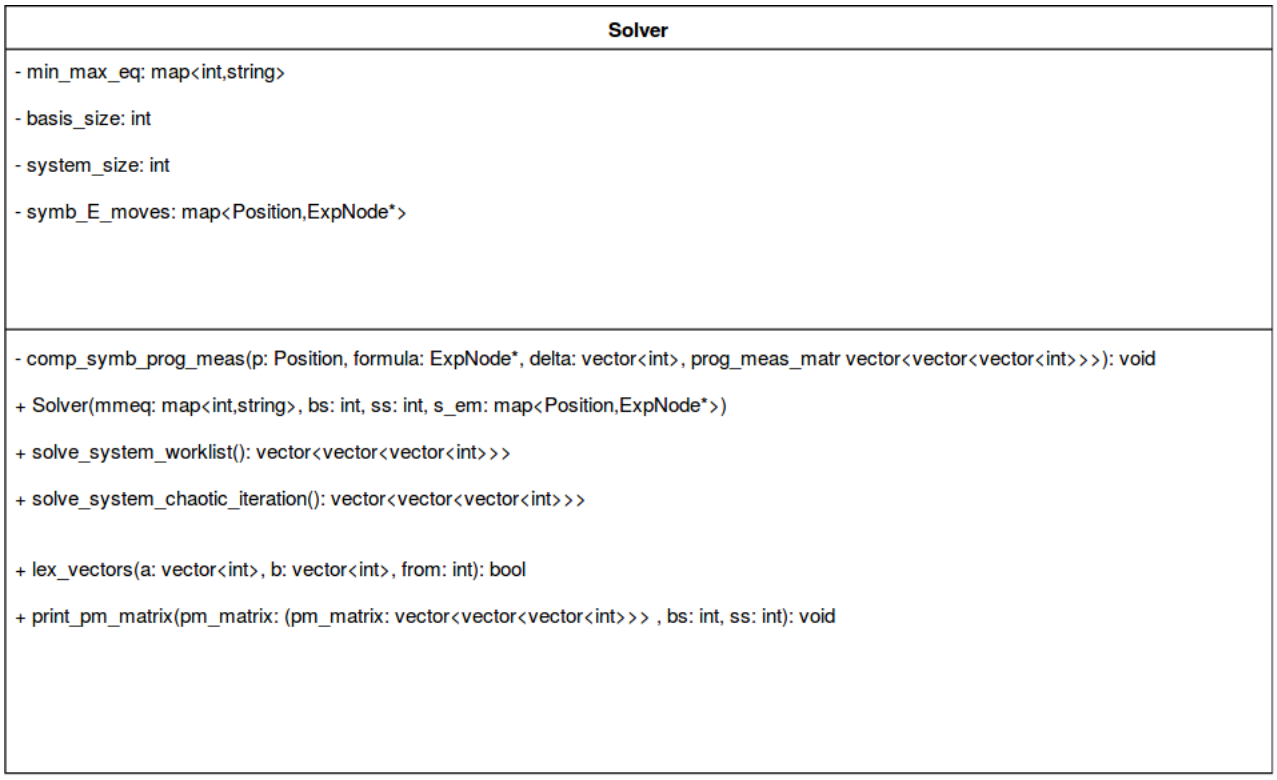
As the name suggests, class `Solver` includes what is needed to solve systems of Progress Measure equations. The following methods are especially important for the class' aims:

* `comp_symb_prog_meas`: this method is used to compute a Progress Measure value on a given position, by means of the inductive definition of Progress Measure in terms of symbolic ∃-moves (see Theorem 3.2.2 of Chapter 3);

* `solve_system_worklist`: it implements an instantiation of the Worklist Algorithm (see [19]) that relies on the strongly-connected components ([6]) of the dependency graph deriving from symbolic ∃-moves (see Chapter 4, Definition 4.1.1);

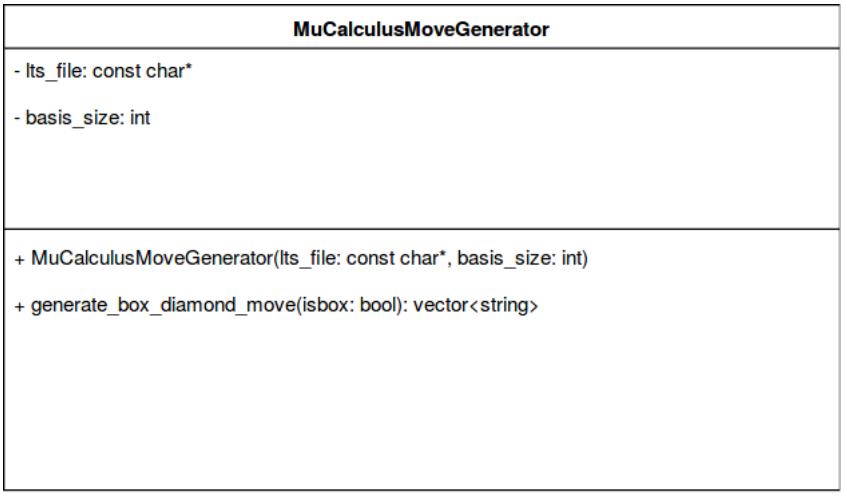
* `solve_system_chaotic_iteration`: implements the Chaotic Iteration algorithm.

  In order to limit the number of iterations, the fixpoint computation halts as soon as any of the components in a given Progress Measure value exceeds the height of the lattice on which the calculation is being performed.



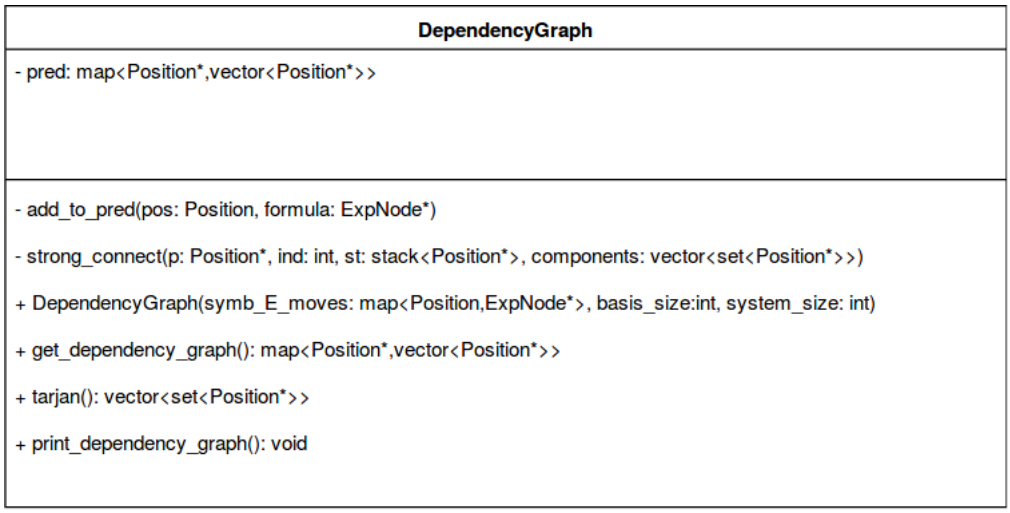

## 5.7 Generating symbolic ∃-moves for the μ-calculus

The procedures implemented inside this class are activated whenever the tool is run in the so-called *μ-calculus* mode (see Appendix A, "User Manual"). Essentially, any instance of this class requires a file encoding a transition system in the *Aldebaran format*, on which it can then produce the symbolic ∃-moves for μ-calculus ◇ and □ operators. Such symbolic ∃-moves (a move for each basis item and for each equation index) are then appended to the system provided by the user and processed by the Move Composer.



## 5.8 The Dependency Graph

`DependencyGraph` represents dependency graphs as adjacency lists. In addition, the couple of methods `strong_connect` and `tarjan` implements Tarjan's algorithm for extracting strongly-connected components ([23]) and topologically sorting the graph.



## 5.9 PGSolver

In order to carry out some experiments with PMModelChecker, we need a tool for handling parity games in a fast way. PGSolver is a collection of tools for generating, manipulating and solving parity games. The aim of this project is to provide a platform for investigating the practical aspects of solving parity games. PGSolver aims at enabling the comparison between different algorithms in terms of their actual performance on various classes of parity games. According to [11], the most recent version of the tool implements the following algorithms from the literature on the subject:

* the recursive algorithm due to Zielonka;
* the local model checking algorithm due to Stevens and Stirling;
* the strategy-improvement algorithm due to Jurdziński and Vöge;
* the strategy-improvement algorithm due to Schewe;
* the strategy-improvement algorithm reduction to discounted payoff games due to Puri;
* the randomised strategy-improvement algorithm due to Björklund and Vorobyov;
* another randomised strategy-improvement algorithm due to Björklund, Sandberg and Vorobyov;
* **the small progress measures algorithm due to Jurdziński**;
* the small progress measures reduction to SAT due to Lange;
* the dominion decomposition algorithm due to Jurdziński, Paterson and Zwick;
* the big-step variant of the latter due to Schewe.

### 5.9.1 Specifying Parity Games

PGSolver has its own specification language for writing and processing parity games, expressed by the following grammar:

⟨ *parity_game* ⟩ ::= [ `parity` ⟨ *identifier* ⟩;] ⟨ *node_spec* ⟩$^+$
⟨ *node_spec* ⟩ ::= ⟨ *identifier* ⟩ ⟨ *priority* ⟩ ⟨ *owner* ⟩ ⟨ *successors* ⟩ [⟨ *name* ⟩];
⟨ *identifier* ⟩ ::= $\mathbb{N}$
⟨ *priority* ⟩ ::= $\mathbb{N}$
⟨ *owner* ⟩ ::= `0` | `1`
⟨ *successors* ⟩ ::= ⟨ *identifier* ⟩ (, ⟨ *identifier* ⟩)$^*$ ⟨ *name* ⟩ ::= " (any ASCII string) "

Since parity games constitute a good source for testing *PMModelChecker*, we wrote a script to translate a parity game written in the above specification into a system of boolean fixpoint equations, using the method described in Chapter 2.

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

**Table 5.2:** A parity game in the PGSolver format.

## 5.10 A real model checker: *mCRL2*

mCRL2 ([29]) is a formal specification language with an associated toolset, developed at the department of Mathematics and Computer Science of the Technische Universiteit Eindhoven. The toolset can be used for modelling, validation and verification of concurrent systems and protocols.
We will use mCRL2 mainly for generating transition systems which can subsequently be accepted by PMModelChecker.

### 5.10.1 The *Aldebaran* format

The Aldebaran format is used to provide specifications of transition systems. The syntax of an Aldebaran file consists of a number of lines, where the first line is the `aut_header` and the remaining lines are referred to as `aut_edge`.

The `aut_header` section is defined as follows:

⟨ *aut_header* ⟩ ::= 'des (' ⟨ *first_state* ⟩ ',' ⟨ *nr_of_transitions* ⟩ ',' ⟨ *n_of_st* ⟩ ')'
⟨ *first_state* ⟩ ::= $\mathbb{N}$
⟨ *nr_of_transitions* ⟩ ::= $\mathbb{N}$
⟨ *n_of_st* ⟩ ::= $\mathbb{N}$

Where:

- *first_state* is a number representing the first state, which should always be 0;
- *nr_of_transitions* is a number representing the number of transitions;
- *n_of_st* is a number representing the number of states.

The `aut_edge` section is defined as follows:

⟨ *aut_edge* ⟩ ::= '(' ⟨ *start_state* ⟩ ',' ⟨ *label* ⟩ ',' ⟨ *end_state* ⟩ ')'
⟨ *start_state* ⟩ ::= $\mathbb{N}$
⟨ *label* ⟩ ::= string
⟨ *end_state* ⟩ ::= $\mathbb{N}$

CHAPTER 6

# Conclusion and Future Work

Our work tackled the model checking problem from a game-theoretic point of view. It is known that the introduction of modal $\mu$-calculus posed the problem of efficient model checking, since $\mu$-calculus formulae involve the solution of nested fixpoint equations. Over the years, research showed that the model checking problem can be converted to a typically game-related issue: finding winning strategies in parity games. One of the most influential techniques related to parity game solution was proposed by Jurdziński. Its seminal research produced the notion of Game Progress Measure, a mathematical tool that can witness to winning strategies in parity games. Jurdziński showed how Progress Measures can be applied to obtain an algorithm which is exponential only in half of the alternation depth of a formula. The importance of Progress Measures became evident, and a large amount of literature has been written on the subject. For instance, [12] extended the notion to general lattices, whereas [2] devised the notion of fixpoint game, and characterised Progress Measures constructively as least fixpoints, making them utilisable in the general context of continuous lattices.

The contribution we proposed is a generic algorithm and a related tool, called *PMModelChecker*, that relies on Progress Measures to solve systems of equations with alternating fixpoints over suitable finite lattices. In particular, we focused on the specific characterisation of Progress Measures developed in [2], which includes the notion of selection, a function that minimises the number of moves considered during the least fixpoint computation, so as to calculate Progress Measures more efficiently. In addition, selection functions are expressed by means of a logical language for upward-closed sets that can concisely represent the moves of each player. Then, [2] uses such logical language in the fundamental concept of symbolic $\exists$-move, which takes part in an inductive definition of Progress Measure. The connection between symbolic $\exists$-moves and Progress Measures is provided by the fact that a symbolic $\exists$-move is given for each game position.

Our tool comprises a set of components that together constitute a complete solver for fixpoint equational systems. We decided not to limit PMModelChecker to the solution of boolean equations and $\mu$-calculus formulae. In fact, any lattice operator appearing in the input system will be customisable by the final user, thus making PMModelChecker profitable in different contexts. The tool's main components are: a customisable parser for systems of fixpoint equations, a composer for basic

symbolic $\exists$-moves, and a solver for Progress Measure equations. We underline the fact that the solver component does not compute directly the initial system's solution. In fact, our solver performs the least fixpoint computation on the system of Progress Measure equations obtained from the set of composite symbolic $\exists$-moves associated with the original system. Essentially, this means that we are taking advantage of a system of least fixpoint equations to solve a system with alternating fixpoints. As a consequence, the final outcome of PMModelChecker's execution should be interpreted according to Progress Measure theory, which states that whenever the Progress Measure value on a game position $(b, i)$ is different from the top value of the Progress Measure lattice, it provides an evidence of the existence of a winning strategy for player $\exists$ in a play starting from $(b, i)$. This fact implies in turn that $b \sqsubseteq u_i$, where $\mathbf{u}$ is the solution vector. We can therefore read the matrix output by our tool as a *yes/no*-valued matrix, where a *yes* value on the cell corresponding to position $(b, i)$ tells us that the inequality $b \sqsubseteq u_i$ is true, whereas, on the contrary, a *no* value provides no evidence for such relation between $b$ and $u_i$. We have already mentioned that Progress Measures have a constructive characterisation as least fixpoints. Because of that, our thesis proposed different methods to carry out such fixpoint computation, ranging from Chaotic Iteration to algorithms employing a worklist. The latter kind of algorithm has been carefully experimented, also by properly exploiting an optimisation described in [6], which relies on the notion of Dependency Graph.

### Future Work

PMModelChecker is still to be considered a prototype, and therefore is subject to several ameliorations, both in terms of usability and computation efficiency.
First of all, a better integration with other existing model checkers, such as mCRL2 or LTSmin, can be experimented. So far, our tool can process (by means of a parser written from scratch) transition systems encoded in the Aldebaran format, which is the old, deprecated format used in the past by the CADP ([27]) toolset. mCRL2 defines other LTS formats (*mCRL2*, *FSM* ([29])) and provides a suitable API for creating and accessing an LTS.

A second issue is related to the fact that putting equations and moves together in the same file looks quite uncomfortable and could cause confusion. Furthermore, each propositional variable that appears in an equation requires its own basic move for each basis item. This could become a problem when dealing with large transition systems.

The third issue that could be tackled is related to the fixpoint computation. The worklist algorithm is only a template that can be instantiated in several manners, and there are plenty of optimisations available in the literature. Since thus far our implementation cannot compete with existing solvers (e.g. parity game solvers) in terms of efficiency, we cannot claim to have achieved a fast fixpoint computation.

APPENDIX A

# User Manual

In this appendix we provide a brief guide on how to download, compile and run *PMModelChecker* on a Linux environment. The guide will terminate with some practical examples.

## A.1 Dependencies

In order to use PMModelChecker you will need the following tools:

* `g++`, from the GNU Compiler Collection (https://gcc.gnu.org/);
* GNU Make (`make`) (https://www.gnu.org/software/make/);
* Flex (`flex`) (the Fast Lexical Analyzer, https://github.com/westes/flex);
* GNU Bison (`bison`, https://www.gnu.org/software/bison/), a general-purpose parser generator;
* Python >2.7 (the Python interpreter, https://www.python.org/).

Moreover, you can utilise the following tools for automatically generating transition systems and parity games, which subsequently will serve as input to PMModelChecker:

* mCRL2, obtainable at https://www.mcrl2.org/web/user_manual/index.html (for transition systems);
* PGSolver, obtainable at https://github.com/tcsprojects/pgsolver (for parity games).

## A.2 Obtaining the sources

You can obtain the source code for PMModelChecker from

https://github.com/Cyofanni/PMModelChecker

Download the latest sources with the following command, from any location on your system:

```
git clone https://github.com/Cyofanni/PMModelChecker
```

The `git clone` command will create a directory named `PMModelChecker`. Then launch

```
cd PMModelChecker
```

The root folder contains the following subdirectories:

* `src/`, which includes the C++ source files;
* `include/`, which includes the header files;
* `scripts/`, which contains some Python scripts written for preprocessing tasks;
* `tests/`, which contains some instances of equational systems, syntax files, parity games and transition systems.

## A.3 Customising the operators

Stay in the project's root directory and define the syntax of the language you will use to write your equational systems, using the format described in Chapter 5. So, let `customsyntax` be the name of the file that includes your language's syntax. Then run

```
python scripts/syntax_generator.py customsyntax
```

After running this command, the couple of files `src/mylex.l` and `src/myparser.yacc` will contain the input needed by Flex and Bison to generate a scanner and a parser for your language.

## A.4 The Preprocessing Phase

The equational system you want to solve must undergo a *preprocessing* phase, so as to speed up the tool's parser. Your system can be preprocessed in the following way:

```
python scripts/preprocessor.py [basis_file] [original_system] \
                        [preprocessed_system]
```

The first command line argument (`[basis_file]`) should be a file containing the names you want to give to the basis elements. Each name should be separated by a newline character. Thus, a possible `basis_file` for a basis of size three must have this content:

```
name_of_item_0
name_of_item_1
name_of_item_2
```

## A.5 Compiling *PMModelChecker*

The root folder of the project contains a *Makefile*, which includes all the instructions needed to compile the final tool. Now run

1. `make parser`
2. `make`
3. `make clean`

## A.6 Running *PMModelChecker*

The `make` command will generate a single executable (in the project's root directory), named `pm_model_checker`, which is to be invoked from the command line in the way explained in the following subsections.

### A.6.1 The *General* Mode

In this mode, the symbolic ∃-move for each simple operator should be provided explicitly by the user.

```
./pm_model_checker -ge [equational_system] [basis_size] \
             [lattice_height] [basis_file]
```

### A.6.2 The $\mu$-*calculus* Mode

When the tool is run in this mode, the symbolic ∃-moves for $\mu$-calculus' $\Box$ and $\Diamond$ operators are computed automatically from a file representing the LTS in the *Aldebaran* format.

```
./pm_model_checker -mu [equational_system] [basis_size] \
         [lts_file] [lattice_height] [basis_file]
```

### A.6.3 The *Normaliser* Mode

PMModelChecker includes a *normaliser*, that is to say, a component that can translate a system of equations containing composite operators into an equivalent (larger) system of normalised equations. The normaliser can be invoked in the following way:

```
./pm_model_checker -normalize [equational_system] [output_file]
```

## A.7 Examples

### A.7.1 Example: a boolean system

A very basic example can be found under the directory `tests/example1`. The file `system_and_moves` defines the system

$$\begin{cases} x_0 =_\mu x_0 \wedge x_1 \\ x_1 =_\nu x_0 \vee x_1 \end{cases}$$

which is a boolean system of two equations with alternating fixpoints, on the lattice `{false, true}`, with basis equal to `{true}`. Evidently its solution is the tuple $[u_0 = false, u_1 = true]$. As you already know, the solver implemented by PMModelChecker relies on Progress Measures defined in terms of symbolic $\exists$-moves. Therefore, any input should contain a move for each operator appearing in the equations, which in our case are

$$\phi^{and}_{true} = [true, 0] \wedge [true, 1]$$

$$\phi^{or}_{true} = [true, 0] \vee [true, 1]$$

So, let us try to run our tool on the system contained in `system_and_moves`, written in the language defined in the file `syntax`. First and foremost, we should generate the specification files needed by Flex and Bison:

```
python scripts/syntax_generator.py tests/example1/syntax
```

Then, from the root directory, launch:

1. `make parser`
2. `make`
3. `make clean`

Now, you need to preprocess the input by means of the following command (from the root directory):

```
python scripts/preprocessor.py tests/example1/basis \
          tests/example1/system_and_moves \
        tests/example1/prep_system_and_moves
```

Now we are ready to solve the system. Observe that in this case we should opt for the *general* mode, because of the fact that we are dealing with a boolean system, and moves for boolean operators have already been supplied. Thus, launch

```
./pm_model_checker -ge tests/example1/prep_system_and_moves \
                  1 2 tests/example1/basis
```

Some helpful notes on the command line arguments:

1. `tests/example1/prep_system_and_moves` contains the preprocessed equational system, i.e. the output generated previously thanks to the script named `preprocessor.py`;
2. `1` is the basis size of the boolean lattice `{false, true}`, which has only one item, `true`. For theoretical reasons, the bottom element of a lattice (which is `false` in our case) cannot be part of the basis;
3. `2` represents the height of the boolean lattice: it is needed for halting the fixpoint computation after a certain amount of iterations;
4. `tests/example1/basis` contains the names of the basis items: it is needed for printing the final result in a readable way.

Some notes on the final output: at the end of the computation, you should see the following lines on the screen:

```
PROGRESS MEASURE MATRIX IS:
[DON'T KNOW (*)][true <= u_1]
```

The so-called "Progress Measure Matrix" (also referred to as `R`) contains all the clues we need in order to infer the structure of the solution. Since in this example we are dealing with a system of two equations over a lattice with a basis of size one, the theory tells us that Progress Measure values will be distributed on a $1 \times 2$ matrix. Rows will be denoted by the corresponding basis items, whereas columns by the zero-based equation indices. Clearly, the matrix output in the current example has one row, corresponding to the only basis item (`true`), and two columns. Now consider each matrix cell:

* `R(true)(0) = [DON'T KNOW (*)]` : we know that whenever $R(b)(i) = *$ (* represents the top of the Progress Measure lattice), we cannot derive any

information on $(b, i)$, i.e. if $\mathbf{u}$ is the solution of the system, we cannot conclude that $b \not\sqsubseteq u_i$. In this case, $b = true$, $i = 0$, and we know from the beginning that $u_0 = false$. Obviously, there is no way to conclude that $true \sqsubseteq false$;

* `R(true)(1) = [true <= u_1]` : here the computation did not reach the top value. Then we can safely infer that $b \sqsubseteq u_1$, where $b = true$ and $u_1 = true$. This is definitely congruent with the fact that $true \sqsubseteq true$.

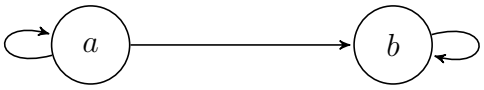
Notice that we could have shown the actual values for each matrix cell (which would have been arrays of naturals), but there was no point doing this, since a matrix of *YES/NO*-like values is fit for our purpose of understanding the solution.

### A.7.2 Example: $\mu$-calculus equations

The directory `tests/example2` contains an example based on $\mu$-calculus. The file `system_and_moves` defines the system

$$\begin{cases} x_0 =_\mu p_0 \vee \Diamond x_0 \\ x_1 =_\nu x_0 \wedge \Box x_1 \end{cases}$$

which is equivalent to the $\mu$-calculus formula $\phi = \nu x_1.((\mu x_0.(p_0 \vee \Diamond x_0)) \wedge x_1)$. We want to verify such formula with respect to the unlabelled transition system depicted below, where the propositional variable $p_0$ is true on the state named $b$ and false otherwise:




The initial system of equations must be solved on the lattice $2^{\mathbb{S}}$, where $\mathbb{S} = \{a, b\}$. The set of singletons $\{\{a\}, \{b\}\}$ constitutes the basis $B_{2^{\mathbb{S}}}$. A specification for this transition system can be found in `tests/example2/transsystem.aut`, which is written in the *Aldebaran* format and can be easily generated with the mCRL2 toolset.
Clearly, the solution to our system is $[u_0 = \mathbb{S}, u_1 = \mathbb{S}]$ (corresponding to the fact that $\phi$ holds in every state).
Now, open the file `tests/example2/system_and_moves` and take a look at the set of moves, which is:

$$\begin{gathered} \phi^{and}_{\{a\}} = [\{a\}, 0] \wedge [\{a\}, 1] \\ \phi^{or}_{\{a\}} = [\{a\}, 0] \vee [\{a\}, 1] \\ \phi^{and}_{\{b\}} = [\{b\}, 0] \wedge [\{b\}, 1] \\ \phi^{or}_{\{b\}} = [\{b\}, 0] \vee [\{b\}, 1] \\ \phi^{p_0}_{\{a\}} = false \end{gathered}$$

$$\phi^{p_0}_{\{b\}} = true$$

Observe that we deliberately omitted the basic moves for the remaining operators, namely, $\Diamond$ and $\Box$, since they will be computed automatically from the transition system when running the tool in the *$\mu$-calculus* mode. Please notice also that propositional variables ($p_0$ in this example) can be considered zero-arity operators with associated trivial basic moves (*true* or *false* atoms, according to the basis item).
Now you can carry out the usual steps.
First, generate the specification files for Flex and Bison:

```
python scripts/syntax_generator.py tests/example2/syntax
```

Then launch the usual build commands:

1. `make parser`
2. `make`
3. `make clean`

Preprocess the input:

```
python scripts/preprocessor.py tests/example2/basis \
          tests/example2/system_and_moves \
        tests/example2/prep_system_and_moves
```

So far, even the preprocessed input (`tests/example2/prep_system_and_moves`) lacks the $\Diamond$ and $\Box$ moves. Now we are ready to launch the tool in the *$\mu$-calculus* mode:

```
./pm_model_checker -mu tests/example2/prep_system_and_moves 2 \
     tests/example2/transsystem.aut 2 tests/example2/basis
```

When run with the `-mu` flag, PMModelChecker reads the transition system specification (`tests/example2/transsystem.aut`), generates the $\Diamond$ and $\Box$ moves and appends them to the equational system file (`tests/example2/prep_system_and_moves`); notice that nothing changes in the non-preprocessed system. We recommend you to preprocess the system again whenever you want to solve it with the *$\mu$-calculus* mode. Let us recap the meaning of each argument:

1. `tests/example2/prep_system_and_moves`, contains the preprocessed equational system, i.e. the output generated previously thanks to `preprocessor.py`;
2. `2` is the basis size of the powerset lattice $2^{\mathbb{S}}$, which has two items, $\{a\}$ and $\{b\}$;

3. `tests/example2/transsystem.aut` is the transition system in the Aldebaran format;
4. `2` is the height of the powerset lattice $2^{\mathbb{S}}$;
5. `tests/example2/basis` contains the names of the basis items.

A quick look at the output (`<=` represents the $\subseteq$ operator):

```
PROGRESS MEASURE MATRIX IS:
    [a <= u_0][a <= u_1]
    [b <= u_0][b <= u_1]
```

Everything is consistent with the known solution, indeed:

* $\{a\} \subseteq u_0 = \{a, b\}$;
* $\{a\} \subseteq u_1 = \{a, b\}$;
* $\{b\} \subseteq u_0 = \{a, b\}$;
* $\{b\} \subseteq u_1 = \{a, b\}$.

### A.7.3 Example: running the normaliser

Under `tests/example3` you can find the file `complex_system`, where, as the name suggests, equations contain composite operators. You may wish to normalise it. Launch the command

```
./pm_model_checker -normalize tests/example3/complex_system \
                tests/example3/normalized_system
```

The file `tests/example3/normalized_system` contains the normalised system, where the right-hand side of each equation has just one operator.

### A.7.4 Example: generating a system from a parity game

Now we show you how to produce a system of fixpoint boolean equations starting from a parity game written in the format used by PGSolver. The game we used to run this example can be found under `tests/example4`.
Firstly, sort the file according to the second column, which stores priorities:

```
tail -n +2 tests/example4/parity_game_1.gm | \
                sort -k 2 -n > \
    tests/example4/parity_game_1_sorted.gm
```

Then, run the converter script (`game_to_system.py`) on the sorted game:

```
python scripts/game_to_system.py tests/example4/parity_game_1_sorted.gm \
                    tests/example4/boolean_system 0
```

The last command line argument could be equal either to 0 or to 1, and has the following meaning:

* 0: we are interested in player 0's win;
* 1: we are interested in player 1's win.

# Bibliography


[1] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, Jiri Srba, 2007. *Reactive Systems: Modelling, Specification and Verification.*
Cambridge University Press.

[2] Paolo Baldan, Barbara König, Christina Mika-Michalski, and Tommaso Padoan, 2019. *Fixpoint Games on Continuous Lattices.*
Proc. ACM Program. Lang.3, POPL, Article 26 (January 2019) https://arxiv.org/abs/arXiv:1810.11404v1

[3] Benerecetti, M., Dell'Erba, D., Mogavero, F. *A Delayed Promotion Policy for Parity Games.*
In: GandALF 2016. EPTCS, vol. 226, pp. 30–45 (2016)

[4] Julian Bradfield and Igor Walukiewicz, 2018. *The mu-Calculus and Model Checking.*
In Handbook of Model Checking, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 871–919.

[5] Julian Bradfield and Colin Stirling, 2005. *Modal Mu-Calculi.*
Editor(s): Patrick Blackburn, Johan Van Benthem, Frank Wolter, Studies in Logic and Practical Reasoning, Elsevier, Volume 3.

[6] Rance Cleaveland, Marion Klein, and Bernhard Steffen, 1992. *Faster model checking for the modal Mu-Calculus.*
In Proc. of CAV 1992 (Lecture Notes in Computer Science), Vol. 663. Springer, 410–422.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest Clifford Stein, 1989. *Introduction to Algorithms.*
The MIT Press.

[8] Patrick Cousot and Radhia Cousot, 1977. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.*
In Proc. of POPL '77 (Los Angeles, California). ACM, 238–252.

[9] Radhia Cousot and Patrick Cousot, 1979. *Constructive versions of Tarski's fixed point theorems.*
Pacific J. Math. 82, 1 (1979), 43–57.

[10] Tom van Dijk, 2018. *Oink: an Implementation and Evaluation of Modern Parity Game Solvers.*
In: Beyer D., Huisman M. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2018. Lecture Notes in Computer Science, vol 10805. Springer, Cham.

[11] Oliver Friedmann, Martin Lange, 2017. *The PGSolver Collection of Parity Game Solvers.*
https://github.com/tcsprojects/pgsolver

[12] Ichiro Hasuo, Shunsuke Shimizu, and Corina Cîrstea, 2016. *Lattice-theoretic progress measures and coalgebraic model checking.*
In Proc. of POPL '16. ACM, 718–732.

[13] Marcin Jurdziński, 2000. *Small Progress Measures for Solving Parity Games.*
In Proc. of STACS '00 (Lecture Notes in Computer Science), Vol. 1770. Springer, 290–301.

[14] Gijs Kant, Jaco van de Pol, 2014. *Generating and Solving Symbolic Parity Games.*
In Proceedings GRAPHITE 2014, arXiv:1407.7671.

[15] Jeroen Keiren, 2009. *Solving Boolean Equation Systems using Small Progress Measures.*

[16] Saul Kripke, 1963. *Semantical Considerations on Modal Logic.*
Acta Philosophica Fennica, 16: 83-94

[17] John Levine, 2009. *flex & bison.*
O'Reilly Media.

[18] Angelika Mader, 1997. *Verification of Modal Properties Using Boolean Equation Systems.*
Ph.D. Dissertation. TU München.

[19] Flemming Nielson, Hanne Riis Nielson, 1999. *Principles of Program Analysis.*
Springer-Verlag Berlin Heidelberg.

[20] Flemming Nielson, Hanne Riis Nielson, 2007. *Semantics with Applications: An Appetizer.*
Springer, Undergraduate Topics in Computer Science.

[21] Dana Scott, 1972. *Continuous lattices.*
In Toposes, Algebraic Geometry and Logic (Lecture Notes in Mathematics), F. W. Lawvere (Ed.). Springer, 97–136.

[22] Perdita Stevens and Colin Stirling, 1998. *Practical Model-Checking Using Games.*
In Proc. of TACAS '98 (Lecture Notes in Computer Science), Vol. 1384. Springer, 85–101.

[23] Robert Tarjan, 1972. *Depth-first search and linear graph algorithms.* In: SIAM Journal on Computing. Vol. 1 (1972), No. 2, P. 146-160.

[24] Alfred Tarski, 1955. *A lattice-theoretical fixpoint theorem and its applications.* Pacific J. Math. 5 (1955), 285–309.

[25] Yde Venema, 2008. *Lectures on the modal $\mu$-calculus.* Lecture notes, Institute for Logic, Language and Computation, University of Amsterdam.

[26] Wieslaw Zielonka, 1998. *Infinite Games on Finitely Coloured Graphs With Applications to Automata on Infinite Trees.* Theoretical Computer Science 200, 1-2 (1998), 135-183.

[27] *CADP: Construction and Analysis of Distributed Processes.* http://cadp.inria.fr/

[28] *LTSmin: Model Checking and Minimization of Labelled Transition Systems.* https://ltsmin.utwente.nl/

[29] *mCRL2: analysing system behaviour.* https://www.mcrl2.org/web/user_manual/index.html

[30] *Oink: an implementation of modern parity game solvers.* https://github.com/trolando/oink