Master in Formal Methods in Computer Science

Univerdidad Complutense de Madrid

Facultad de Informática

Calificación: 7.5

The Temporal Booleanization Theorem: realizability checking over numerical-LTL industrial requirements

---

Author

Andoni Rodríguez

2020/2021

Master in Formal Methods in Computer Science

Univerdidad Complutense de Madrid

Facultad de Informática

Calificación: 7.5

The Temporal Booleanization Theorem: realizability checking over numerical-LTL industrial requirements

---

Author

Andoni Rodríguez

Supervisor

César Sánchez

Acknowledgments

First of all, I would like to express my gratitute to mi supervisor: César Sánchez. César has not only helped me with the thesis and corrected it with great care and attention, but he has also taken the time to facilitate my inclusion in such a difficult and technical world as formal methods. He has made me a better researcher and I will always be grateful for that. And I believe that there is no better compliment.

I would like to thank the Master itself and its universities for offering such an interesting formation and its professors for having facilitated the course so much in this atypical year. Also, I also cannot forget Paqui Lucio, thanks to whom I was able to get to know both IMDEA and the Master's program itself, and who I will always consider a very appreciated person.

In addition, I would like to remember some students from the Master's which whom I have had the opportunity to share a coffee or some burritos: Jurjo, Víctor, Álex, Roland, Chema, Matías.

And of course I would like to thank my family.

Last but no least I would like to thank all the scientific and formal effort made by lots of people before me that has opened a wonderful study field that, thanks to them and other technologies, is nowadays at the crest of the innovation wave.

My journey in the world of research has only just begun. And I think this acknowledgement letter will be much longer in a very few months.

Abstract

Industrial systems are getting more complex every year, and due to that complexity growth, the languages to specify them are becoming increasingly more expressive: so that they can properly model both controllable parts and the environment, or even temporal events.

As a result, a great effort has been made over the last few years to move forward in the area of realizability checking of complex requirements.

However, there is a blocking problem in the state-of-the-art realizability checkers: they only accept requirements that only contain Boolean variables on them. Therefore, these checkers cannot handle many real-industrial requirements, like those requirements containing numerical variables.

One approach is to *Booleanize* numeric requirements and convert them into equivalent Boolean requirements.

This problem has been researched and it has been discovered that its solution is not trivial. Thus, the main contribution of this thesis is that we have proved a theorem which verifies that (1) a correct Booleanization exists for all requirements that use theories with a decidable $\exists^*\forall^*$ fragment; and, thus (2) we have a realizability checking procedure for numeric requirements based on safety two-player turn-based LTL games.

In addition, an algorithm that performs this has been proposed and implemented in OCaml.

Resumen

Los sistemas industriales son cada vez más complejos, y debido a ese crecimiento de complejidad, los lenguajes de especificación son cada vez más expresivos: de forma que puedan modelar adecuadamente tanto las partes controlables del sistema como su entorno; o incluso eventos temporales.

Por ello, en los últimos años se ha hecho un gran esfuerzo por avanzar en el ámbito de la comprobación de la realizabilidad de los requisitos complejos.

Sin embargo, existe un problema limitante en los comprobadores de realizabilidad de estado del arte: sólo aceptan requisitos que únicamente contengan variables booleanas en ellos. Por lo tanto, no pueden manejar muchos requisitos de la industria real, como los que contienen variables numéricas.

Una solución es convertir todos los predicados que contengan variables no booleanas (es decir, *booleanizarlos*), creando unos requisitos que sean equi-realizable y puramente booleanos.

Se ha investigado este problema y se ha descubierto que su solución no es trivial. Así, la principal contribución de la presente tesis es que se ha demostrado un teorema que verifica que (1) existe una booleanización correcta para todas las teorías numéricas cuyo fragmento $\exists^*\forall^*$ sea decidible; y, por tanto (2) tenemos un procedimiento de comprobación de realizabilidad para juegos LTL safety de dos jugadores escritos numéricamente.

Además, se ha propuesto un algoritmo que realiza esto, y se ha implementado en OCaml.

Keywords

0.1. Keywords

Booleanization, Realizability, Linear Temporal Logic, Infinite Games, Quantifier Elimination, Requirement Synthesis.

0.2. Palabras clave

Booleanización, Realizabilidad, Lógica Temporal Lineal, Juegos Infinitos, Eliminación de Cuantificadores, Síntesis de Requisitos.

General index

Acknowledgments I

Abstract III

Resumen V

Keywords I

0.1. Keywords I

0.2. Palabras clave I

General index III

Figure index VII

Table index IX

1. Introduction 1

1.1. General framework 1

1.1.1. An extended abstract 1

1.1.2. Motivation 2

1.2. Proposed solutions 3

1.2.1. What is to Booleanize Correctly 3

1.2.2. How to Booleanize Correctly . . . 3
1.3. About the thesis . . . 4
1.3.1. Objectives . . . 4
1.3.2. Thesis' distribution . . . 4

2. Theoretical Foundations **5**
2.1. Requirements, Temporality and Reactivity . . . 5
2.1.1. Industrial requirements . . . 6
2.1.2. Modal logics and Temporal logics . . . 11
2.1.3. Linear Temporal Logic . . . 15
2.1.4. Reactive Systems: Realizability vs Satisbiability . . . 21
2.2. Infinite games, Realizability and Synthesis . . . 25
2.2.1. Games, Arenas, Plays and Strategies . . . 26
2.2.2. Reachability Games and Safety Games . . . 30
2.3. Decision Procedures: Quantifier Elimination . . . 33
2.3.1. Basics on quantifier elimination . . . 34
2.3.2. The Fourier-Motzkin algorithm . . . 36
2.3.3. Test-points and quantifier elimination . . . 41

3. The Booleanization Theorem **43**
3.1. Wrong ideas, theorems and algorithm . . . 43
3.1.1. Final solution: A global Booleanization is possible . . . 43
3.1.2. Brute-force Booleanization Algorithm . . . 54
3.1.3. An incorrect method: using queries . . . 60
3.2. Correctness Proof of the Correct Algorithm . . . 64
3.2.1. The Local Booleanization Theorem . . . 64
3.2.2. Temporal Booleanization Theorem . . . 67

4. Conclusions **99**

4.1. Future Work 99

4.2. Two parallel researches 100

4.2.1. Complexity analysis and heuristics 100

4.2.2. Fast algorithms 101

4.3. Upcoming Empirical Analysis 102

Appendices

A. Requirements vs Requisites **105**

Bibliography **107**

Figure index

2.1. The classic Waterfall model with its phases. 6

2.2. A snapshot for an Expert System in gastrointestinal diseases 11

2.3. Representation of the Globally LTL operator. 16

2.4. Representation of the Finally LTL operator. 16

2.5. Representation of the Next LTL operator. 17

2.6. Representation of the Until LTL operator. 17

2.7. Representation of the Release LTL operator. 18

2.8. Different power of a player depending on the quantification of its v_1 to v_n variables. 23

2.9. Different power paradigms in a two-player system: the purple colour area indicates the realizability (see Definition 2.2), while the green one shows different levels of system´s power; in addition, the satisfiability (see Definition 2.1), the realizability and the *hardest consistencies'* (i.e. all the variables are universally quantified) points are indicated. 24

2.10. Expressivity of the specification that we are generally talking about: LTL + reactivity. 25

2.11. Graphical example for an arena. 27

2.12. A sub-arena of the arena in Figure 2.11 27

2.13. A play over the arena in Figure 2.11 28

2.14. An arena of a reachability game. 31

2.15. An arena of a safety game. 33

3.1. The diagram of the SMT Library, with some modules that solve quantified formulae. 51

3.2. A diagram with timesteps, where we can see all the ψ subformulae (each point) are timesteps previous to the φ. 68

3.3. A tree-form LTL specification with a given trace, where the red points are subformulae and where the subformulae withing the green area denote those subformulae that fall off the trace (i.e. whose literals are all true). . 71

3.4. An arbitrary numeric game and its equivalent Boolean game, and the relation between their positions. 83

3.5. A simplified graph, where the squares denote environment positions and the round denote system positions (that is, it does not show the needed information that each positions carries -see Definition 3.18) and where red colour denotes unrealizability, contrary to the green colour. Also, double arrows mean potentially infinite transitions. 84

3.6. A version of Figure 3.5 where the history bit has been applied, producing a different realizability result. 84

3.7. A Booleanized and numeric arenas' frames, and the relationship between some states. 86

3.8. An arbitrary p position of the Boolean game is related with an arbitrary q position of the numeric game; and their post-positions are also related. . . 87

3.9. A visual representation of the structural proof of Lemma 3.5, where k is greater or equal than the temporal depth (see Definition 3.13). 91

3.10. A visual representation of proof of Lemma 3.6, using the same construction of Example 3.12. Here, $k \geqslant 0$ is an arbitrary trace length (see Definition 3.12), while '$\cdots$' denotes that a set of logic operators (whether temporal or not) have been used or not. 92

4.1. The diagram of how the Forwards Pass of Cooper's Booleanization (thus, also MiniCooper) algorithm has been implemented in Ocaml. 101

A.1. Comparison between languages on how to say *requirement* and requisite . 105

Table index

2.1. Different types of modal logics, their most paradigmatic symbols and meaning of those symbols. 12

List of Algorithms

1. Brute-force Booleanization algorithm 52
2. The query-based Booleanization method as an algorithm 61

1. CHAPTER

Introduction

In this first chapter we will give a general overview of the thesis, together with the first technical insights.

1.1. General framework

First of all, since the abstract offered in English (and in Spanish) is as concise and straightforward as possible, we offer an extension of it in Subsection 1.1.1 below, which sets the context of the thesis and its contribution somewhat more clearly.

1.1.1. An extended abstract

The main purpose of this master's thesis is to solve a Formal Methods' problem that leads to a really clear practical use in the industry. Concretely, industrial systems are formally described or, more properly said, *specified*, using requirements (we are talking about functional requirements -see Subsubsection 2.3-). Depending on the logics used for those requirements, the described system can be more or less expressive.

The requirement synthesizing state-of-the-art tools are getting enhanced year by year, and nowadays we can even synthesize complex systems like two-player games with temporality. For instance: if *Player 1* performs the *a* action, then *Player 2* has to perform the *b* action in less than *n* time units.

However, it is well-known that some sets of requirements are not *synthesizable* or, properly said, the requirements are not *realizable*. That happens due to possible inconsistencies (like contradictions) on these requirements; in the same way that there are unsatisfiable propositions in propositional logic .

To check whether a specification is or not realizable, there are another state-of-the-art tools called realizability checkers. One problem is that those tools (such as *AbsSynthe* [Brenguier et al., 2014]) only accept requirements that contain exclusively Boolean variables on them. Therefore, they cannot handle many real requirements, like those requirements containing numerical variables. For instance: *if* $(x < y)$ *then* z, where $x \in \mathbb{R}$, $y \in \mathbb{R}$ and $z \in \mathbb{B}$, note that someone could invent a realizability checker that handles numerical variables. One approach is to Booleanize, particularly if we wish to use AbsSynthe.

This problem has been researched and it has been discovered that its solution is not trivial, and does not only depend on simple strategies like queries to a solver or bitwise approximation. This is due to the fact that we have to design a method that (1) not only Booleanizes the predicates, but that also (2) preserves the dependencies between the original variables, as well as (3) decides which variable-owner or *player* (if requirements model a game) is the owner of each substitution fresh Boolean variable.

The main contribution of this thesis is a theorem which captures that (1) a correct Booleanisation exists for all numeric theories with a decidable $\exists^*\forall^*$ fragment; and, thus (2) we have a realizability checking procedure for numeric-written safety two-player turn-based LTL games (and, therefore: also for their synthesis).

In addition, an algorithm that does this has been proposed (and implemented in OCaml), as well as two substantial improvements to it, based on Cooper's and Ferrante-Rackoff's quantifier elimination methods.

To follow the formal-practical purpose of this thesis, we offer these algorithm's correctness theorem, as well as the algorithms themselves and some empirical cases' evaluation with real industrial requirements.

1.1.2. Motivation

The motivation of this thesis is twofold:

- A scientific motivation: the need of a theoretical correct method to allow numerical LTL realizability.

- A practical motivation: the industrial need to verify numeric written LTL specification (and ultimately, synthesize them).

1.2. Proposed solutions

1.2.1. What is to Booleanize Correctly

The realizability checking (see Section 2.1.4) of reactive numeric LTL requirements (see Subsection 2.1.3) is not possible by the state-of-the-art tools, so one possible solution is to *Booleanized* the requirements.

Booleanizing a set of requirements consists of replacing all numeric literals (i.e. predicates) by Boolean variables.

When the requirements model a two-player *environment vs system* game, this problem is additionally challenging.

The main issue is preserving the exact original power of each player or variable-owner of the numeric game, which is challenging (see an example in Subsubsection 3.1.3 below).

Correctly Booleanizing is to translate the specification from numeric to Boolean, while maintaining these mentioned powers.

1.2.2. How to Booleanize Correctly

Some obvious method such as the bitwise Booleanization [Losada, 2020] is not applicable, since can be used only with enumerations.

Some other tempting methods such as the query based Booleanization (see Subsection 3.1.3), propose to Booleanize via SMT queries over the inequalities. But these simple methods are shown to be incorrect.

Our solution in Section 3.1.1, proposes to (1) give the ownership of all the Booleanized predicates to the one player and then (2) managing the dependencies between the Boolean variables by computing a new requirement called the *extra requirement* (see Definition 3.10). This computation is also made with validity queries, but in a different way, which has been proven correct in Section 3.2.

1.3. About the thesis

1.3.1. Objectives

There are three goals:

1. Study the state of the art in Booleanization techniques.
2. Provide a formally correct method for Booleanization of reactive requirements.
3. Implement the Booleanization method proposed in the OCaml language.

1.3.2. Thesis' distribution

The thesis consists of four chapters, plus a small annex.

1. Introduction (see Chapter 1): This chapter contains an introduction to the thesis.
2. Theoretical Foundations (see Chapter 2): The theoretical bases necessary to understand both the state of the art or the research itself are detailed.
3. The Booleanization Theorem (see Chapter 3): This chapter proposes a solution in the form of an algorithm and its implementation, and a correction theorem.
4. Conclusions (see Chapter 4): We raise the impact of the work, parallel research lines and future work.

2. CHAPTER

Theoretical Foundations

We introduce in this chapter the most relevant fields and concepts for this research in a formal way, as well as their underlying concepts that will later be needed contributions to fully understand the development of this thesis, both in the state of the art and research contributions.

2.1. Requirements, Temporality and Reactivity

The original problem of the thesis comes from the fact of having to work with industrial requirements, that describe a controller (i.e. the *system*) having to react to a hostile agent (i.e. the *environment*), that is, a reactive system. Furthermore, the requirements include temporal constraints, typically expressed in temporal logic.

To sum up: we have (1) some industrial requirements, (2) which are then modeled as a reactive system (3) using temporal logics.

Let us see here what these three concepts are and how they are related.

2.1.1. Industrial requirements

Requirements Engineering

There are many industrial engineering guidelines for software and hardware production, such as the waterfall model [1], which breakdowns project activities into linear sequential phases, where each phase depends on the deliverables of the previous: requirements, design, implementation, verification and maintenance, as it is depicted in Figure 2.1.

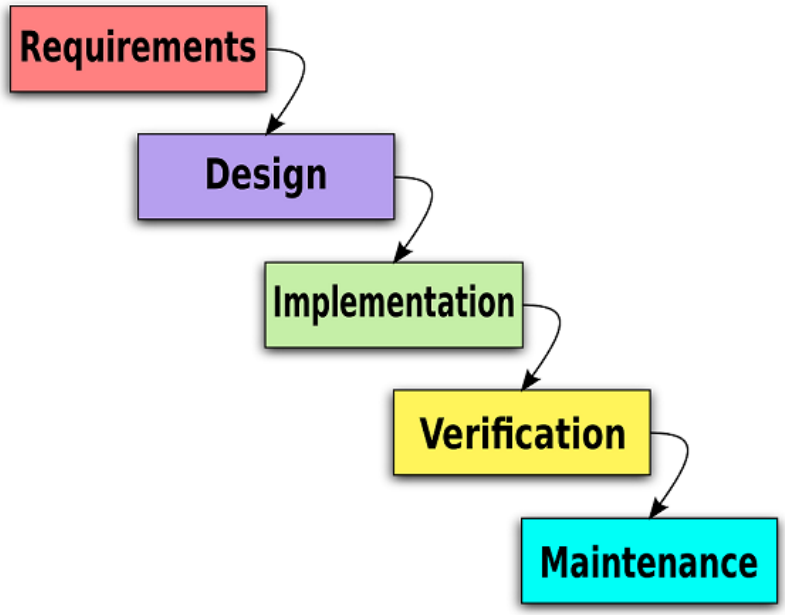

Figure 2.1: The classic Waterfall model with its phases.
Source: https://www.seowebsitedesign.com/the-waterfall-model-of-software-development/

Even if some later development methods (such as the Rational Unified Process for software [2]) assume that requirements engineering continues through a system's lifetime, in all of the industrial guidelines, there is a common factor: requirements engineering is presented as the first phase of the development process.

Description **2.1.** *In systems engineering and software engineering, requirements engineering is the process of defining, documenting, and maintaining requirements in the engineering design process.*

[1] https://en.wikipedia.org/wiki/Waterfall_model
[2] https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

In other words, the waterfall method comprises all tasks related to the determination of the needs or conditions to be satisfied for new or modified software or hardware, taking into account the various requirements of stakeholders, which may conflict with each other.

The activities involved in requirements engineering vary widely, depending on the type of system being developed. These may include:

1. Requirements gathering: Developers and stakeholders meet; the latter are inquired concerning their needs and wants regarding the system.
2. Requirements analysis: Requirements are identified and conflicts with stakeholders are solved. Both written and graphical tools are used as aids. Examples of written analysis tools are use cases, whereas examples of graphical tools are UML diagrams.
3. Requirements specification: Requirements are documented in a formal artifact called a *requirements Specification*, which will become official only after validation. An example is the software requirements specification.
4. Requirements verification and validation: Checking that the documented requirements and models are consistent and meet the stakeholder's needs. Only if the final draft passes the validation process, the requirements specification becomes official.
5. Requirements management: Managing all the activities related to the requirements since requirement gathering, supervising as the system is developed, and even until after it is put into use (e. g., modifications, extensions, etc.).

From these activities, Formal Methods research is usually interested in *specification* and in *validation* and *verification*.

Requirement specification

Even if it is usually wrongly used, what a requirement specification is, is well defined:

Description **2.2.** *A requirements specification is a description of a system to be developed.*

The characteristics of a recommendable requirement specification are defined by the IEEE 830-1998 standard [3]:

[3] https://standards.ieee.org/standard/830-1998.html

- Complete: All requirements must be reflected in it and all references must be defined.
- Consistent: It must be coherent with the requirements themselves and also with other specification documents.
- Unambiguous: The wording must be clear so that it cannot be misinterpreted.
- Correct: The system must meet the requirements of the specification.
- Traceable: There has to be possibility of verifying the history, location or application of an item through its stored and documented identification.
- Prioritisable: It must be possible to organise requirements hierarchically according to their relevance to the business and classify them into essential, conditional and optional.
- Modifiable: Although all requirements are modifiable, they should be easily modifiable.
- Verifiable: There must be a finite no-cost method to test it.

In addition, requirements specification lays out functional and non-functional requirements, the difference between which is essential to know.

We define what a functional requirement is in Description 2.3 below.

Description **2.3.** *Functional requirements define a function of a system or its component, where a function is described as a specification of behavior between outputs and inputs. Functional requirements may involve calculations, technical details, data manipulation and other specific functionality that define what a system is supposed to accomplish.*

Whereas we define what a non-functional requirement is in Description 2.4 below.

Description **2.4.** *Non-functional requirements specify criteria that can be used to judge the operation of a system, rather than specific behaviors. They impose constraints on the design or implementation: such as performance requirements, security, or reliability.*

So, in other words, functional requirements are expressed in the form *system must do <requirement>*, while non-functional requirements take the form *system shall be <requirement>*.

We will refer to the first description, Description 2.3, whenever we use the word *requirement* on its own.

Requirements validation and verification

On the other side, we have requirements validation and verification:

Description **2.5.** *Requirements verification and validation is the process of checking that a system meets specifications and that it fulfills its intended purpose.*

Each of these activities responds to a different point questions:

- Verification: Are we building the system right (i.e. assuming we should build X, does our software achieve its goals without any bugs or gaps)?
- Validation: Are we building the right system (i.e. was X what we should have built, or, more formally: does the product conform to the specifications)?

The first question is the key question in the *Formal Verification* area [4], together with the *consistency* attribute of the a specification.

What is truly relevant for the thesis is not how requirements are used or integrated in production pipelines, but the requirements themselves. And more precisely: how requirements are described or *specified*, that is, which is the *language* used to do so.

Let us move to formally specify the requirement's languages.

Requirements' specification language

For many decades industrial systems have been specified using natural-language-written requirements (remember, we are talking about functional requirements), so that they are readable for everyone; even people with no formal knowledge.

However, this approach has many drawbacks if we compare it with formal approaches. Let us review three of them:

[4] https://en.wikipedia.org/wiki/Formal_verification

- No complete abstraction: When we use natural language it is easy to fall into too many unnecessary concretions about the described system, i.e. we usually do not abstract the system as much as we could using formal languages.

- Language ambiguity: Using natural languages, each engineer can specify the system in a different manner, and this way there is not a standard, generating some problems like the difficulty of retaking a system of sharing it with others.

- Lack of expressivity: Once the described system's expressivity gets increased, it gets harder to specify it with natural language.

Due to this, in lots of contexts, industrial systems are described using formal specifications, i.e. using logics to describe behaviour. This way, the mentioned three problems are desirably overcome: there is complete abstraction, there is complete standardization and the expressivity is accurately measured.

For instance, if the system is a closed system (i.e. a function, a.k.a a *transformational system*, or a *sequential system*), that is, that receives input, performs computations and returns output, then classical logics are enough. Sometimes propositional logics can be enough; this usually happens in expert systems [5].

Consider the example snapshot in Figure 2.2, which is a case based expert system to diagnose gastrointestinal diseases. There, we could abstract -capture- all the questions into Boolean variables and have an implication tree.

Whereas in other situations, we must, for instance, use non-Boolean variables (i.e. predicates) and even quantify over the variables appearing in the variables.

Whether these logics are decidable or not is not the point so far, and indeed it will be discussed later in Subsection 2.1.4. For the time being, we only talk about them as purely descriptive languages.

However, the type of industrial systems in which we are interested in this thesis are ones whose relevance is growing day by day: the reactive systems. So let us see what they are and after that, how they are specified using *time*.

[5] https://en.wikipedia.org/wiki/Expert_system

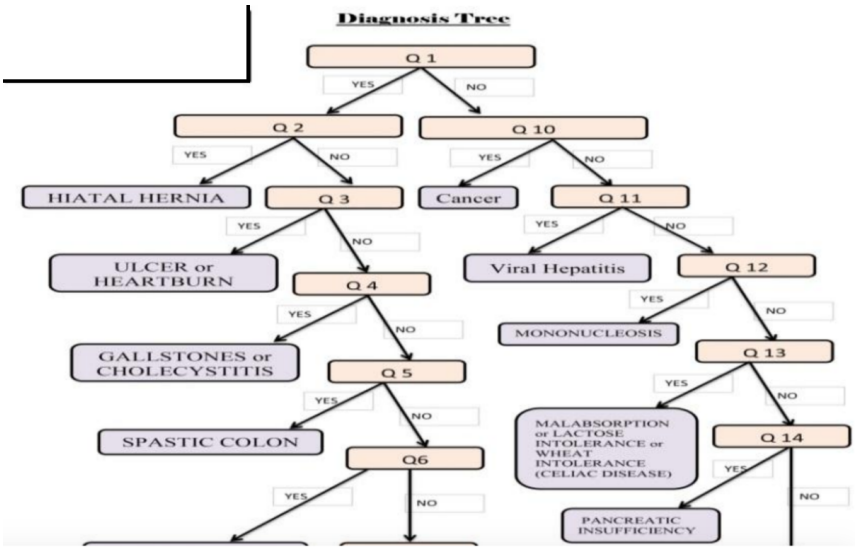

Figure 2.2: A snapshot for an Expert System in gastrointestinal diseases

Source: https://www.researchgate.net/figure/Decision-tree-of-the-diagnosis_fig15_319208260

2.1.2. Modal logics and Temporal logics

One could want to extend the specification logic with other operators and connectives in order to express different ideas. For instance, we could want to add the *maybe*, which happens in epistemic logic [6].

This is what modal logic offers to us.

Basics of modal logics

A modal is an expression that is used to qualify the truth of a judgement. Modal logic is, strictly speaking, the study of the deductive behavior of the expressions *it is necessary that* and *it is possible that*, which is more formally defined in Description 2.6 below.

Description **2.6.** *[Modal logic] Modal logic is a set of formal systems used to represent necessity (represented with the symbol* $\Box$*) and possibility (represented with the symbol* $\Diamond$*).*

However, the term *modal logic'* is used more usually to cover a family of logics with

[6] https://en.wikipedia.org/wiki/Epistemic_modal_logic

Logic	Symbols (some)	Meaning
Deontic logic	Op	It is obligatory that p
	Pp	It is permitted that p
	Fp	It is forbidden that p
Epistemic logic	$K\ x\ p$	x knows that p
Doxastic logic	$B\ x\ p$	x believes that p

Table 2.1: Different types of modal logics, their most paradigmatic symbols and meaning of those symbols.

similar rules and a variety of different symbols. We can see some of these logics in Table 2.1.

The most familiar logics in the modal family are constructed from a weak logic called K (because of Saul Kripke). Each of these logics are well-defined (define their well-formed formulae) and share some properties, such as Lemma 2.1 below.

Lemma **2.1.** *[Necessitation Rule] Let K be the Kripke logic, then:*

$$\textit{If } A \textit{ is a theorem of } K, \textit{ then so is } \Box A$$

In addition, each modal logic contain its own properties. For instance, in a classical modal logic, each $\Box$ and $\Diamond$ can be expressed in terms of the other and negation in a duality law:

$$\Diamond P \leftrightarrow \neg\Box\neg P, \qquad \Box P \leftrightarrow \neg\Diamond\neg P$$

The modal logic in which we are interested in this thesis is temporal logic (see the immediately next Subsubsection 2.1.2).

Temporal logics: an overview

Temporal Logic (see Description 2.7 below) is used to cover formal reasoning about time and temporal information within a logical framework. For instance, representing statements (and differences between statements) like *I am always hungry*, *I will eventually be hungry* or *I will be hungry until I eat something*.

Description **2.7.** *[Temporal logic] Temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time*

The standard semantics of TL is essentially a Kripke-style semantics from modal logic (whose study is out of the scope of this thesis), but incorporating Prior's tense logic to it, which extends the standard propositional language (with atomic propositions and truth-functional connectives) by four temporal operators with intended meaning as follows:

- P: *It was the case that p* (where P stands for 'past').
- F: *It will be the case that p* (where F stands for 'future').
- G: *It always will be the case that p* (where G stands for 'globally').
- H: *It always was the case that p.*

Note that G (for the future), and H (for the past) are the minimal operators of this logic´s syntax together with the propositional symbol's minimal operators (see *-functional completeness* in classic mathematical papers like [Post, 1941], [Wesselkamper, 1975] or [Massey, 1975]).

Also note that these temporal operators can be combined if we let ρ be an infinite path. This way we can model property-satisfaction over infinite traces (see Subsection 2.1.4 and Section 2.2), like ikustrated in Example 2.1 below.

Example **2.1.** *[Temporal logic combinations] Two very typical combinations in specification using temporal logics.*

For instance, It will always be the case that Prior invented Tense Logic translates in TL as $GP(\text{Prior invents TL})$ and has the formal reading: It will always be the case that it has at some time been the case that Prior invents Tense Logic.

This way, we can define more combinations. Let ϕ be a formula which may or may not be temporal. Then:

- $\rho \models FG\phi$*: 'At a certain point, ϕ is true at all future states of the path'.*
- $\rho \models GF\phi$*: 'ϕ is true at infinitely many states on the path'.*

Note that the order of the operators is relevant.

Specifying such infinite traces is one of the applications of temporal logic, which also include (1) its use as a formalism for philosophy that deals with time, (2) a language for representing temporal knowledge in artificial intelligence, among others.

However, we are interested in the application in formal verification, where it is used to state requirements of hardware or software systems (see Example 2.2 below).

Example **2.2.** *[Industrial specification using using temporal logic]*

Let a specification be composed of two requirements:

- R_0*: It is always the case that if '*$STATE == Start\_Heater$*' holds, then '*$STATE == Warmup$*' holds.*
- R_1*: It is always the case that if '*$(STATE == STATE\_Warmup)$ *and* $(CurrentTemp < DesiredTemp)$*' holds, then* $\neg(STATE == STATE\_Warmup)$ *and* $(ShowWarmupTemp == true)$ *hold.*

Both R_0 *and* R_1 *can be formalized in temporal logic, with these formulae:*

- $R_0 : \varphi_0 = G(STATE == Start\_Heater \implies STATE == Warmup)$
- $R_1 : \varphi_1 = G((STATE == STATE\_Warmup) \wedge (CurrentTemp < DesiredTemp) \implies \neg(STATE == STATE\_Warmup) \wedge (ShowWarmupTemp))$

Then, if the specification wanted the conjunction of both requirements, the global specification will be:

$$\varphi = \varphi_0 \wedge \varphi_1$$

Some modalities of temporal logic allow time-branching, meaning that its model of time is a tree-like structure in which the future is not determined and instead there are different paths in the future, any one of which might be an actual path that is taken in a given run.

A paradigmatic example of these branching logics is the Computational Tree Logic (CTL) by [Clarke and Emerson, 1981], which incorporates, among others, two new operators:

- $A\phi$: ϕ has to hold on all paths starting from the current state (A stands for 'all').
- $E\phi$: there exists at least one path starting from the current state where ϕ holds (E stands for 'exists').

CTL is used in formal verification of software or hardware (typically using model checkers -for instance, see *bounded model checking* in [Audemard et al., 2002]-) , to express properties such as *safety* (see Subsection 2.2.2) -i.e. CTL can specify that when some initial condition is satisfied then all possible executions of a program avoid some undesirable condition.

However, branching modalities (such as CTL itself), can be too complex to use in real cases, such as in the industry. Therefore, the fragment of temporal logics in which we are interested is the Linear Temporal Logic (see Description 2.8).

2.1.3. Linear Temporal Logic

Linear Temporal Logic, described in Description 2.8 below, was first proposed for the formal verification of computer programs in 1977 by [Pnueli, 1977]. It is nowadays widely used to describe specifications in object behavior, cooperative protocols, runtime verification, reactive protocols, reactive systems, digital circuits, concurrent programs and, in general, to reason about dynamic systems whose states change over time.

Description **2.8.** *Linear temporal logic or linear-time temporal logic (also known as LTL) is a modal temporal logic with modalities referring to linear time.*

In LTL, one can encode formulae about the future or past of single paths, like: *a condition will eventually be true* or *a condition will be true until another fact becomes true*. LTL allows expressing future-only, past-only formulas or a combination of future and past.

LTL formulae are evaluated on runs, i.e. on infinite linear sequences of states:

$$s[0] \to s[1] \to \ldots \to s[t] \to \ldots$$

In terms of expressive power, LTL is a fragment of the classic first-order logic [7]. Also, model checking and satisfiability against an LTL formula are PSPACE-complete problems. LTL synthesis (for instance, see [Camacho et al., 2018]) and the problem of verification of two-player games (see Section 2.2) against an LTL winning condition is 2EXPTIME-complete.

[7] http://www.lsv.fr/~gastin/Verif/DiekertGastin-FO-07.pdf

LTL Syntax and semantics

LTL provides both the unary and the binary the operators of propositional logic: $\neg$, $\wedge$, $\vee$, $\Longrightarrow$, $\leftrightarrow$. It also includes temporal connectives.

LTL provides the following future operators: $\Box$, $\Diamond$, $\circ$, $\mathcal{U}$, $\mathcal{R}$,$\mathcal{W}$. Their writing standard and informal semantics are given below:

- Unary operators:
  - $G\phi$ or $\Box p$ or **G**lobally: It is used to express invariants: that is, a property ϕ that holds in all future states. See Figure 2.3 below.

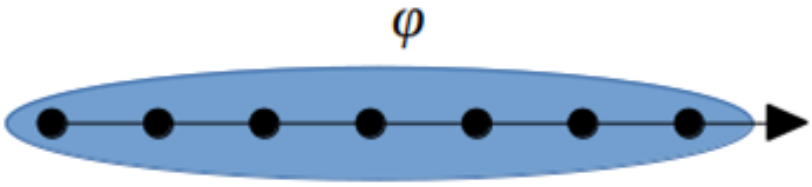

Figure 2.3: Representation of the Globally LTL operator.

Source: [Losada, 2020]

  - $F\phi$ or $\Diamond\phi$ or **F**inally: It is used to express a property that must eventually hold at some unknown point in the future. See Figure 2.4 below.

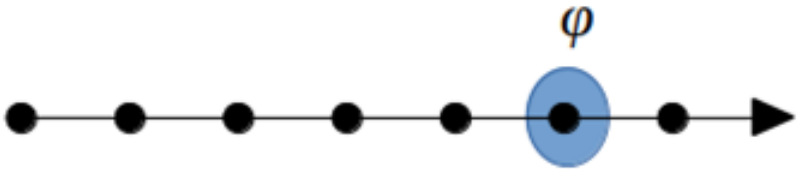

Figure 2.4: Representation of the Finally LTL operator.

Source: [Losada, 2020]

  - $\mathcal{X}\phi$ or $\circ\phi$ or Ne**X**t: It is used to describe a property that must hold in the next state of the computation. See Figure 2.5 below.
- Binary operators:

Figure 2.5: Representation of the Next LTL operator.

Source: [Losada, 2020]

- $\psi \mathcal{U} \phi$ or **U**ntil: It is used to express that some property ϕ holds up to the point when other property ψ holds. See Figure 2.6.

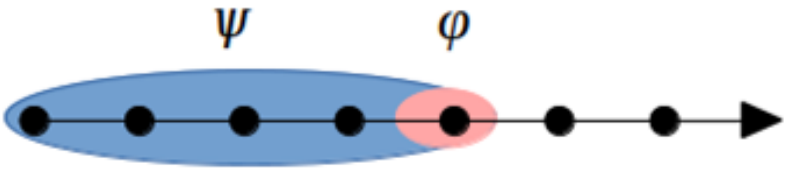

Figure 2.6: Representation of the Until LTL operator.

Source: [Losada, 2020]

- $\mathcal{W}$ is very similar to $\mathcal{U}$, but it does not require q to be satisfied, in which case ϕ is allowed to hold forever.
- $\phi \mathcal{R} \psi$ or **R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever. See Figure 2.7.

All these operators have their *past counterparts* (see Subsubsection 2.1.3).

Note that with these operators, we can now specify different properties. For instance, let us consider a use case similar to Example 2.2 in Example 2.3 below.

Example **2.3.** *[Industrial specification using using LTL]*

Let a specification be composed of two requirements:

- R_0*: It is always the case that if 'STATE == Start_Heater' holds, then 'STATE == Warmup' holds after at most* 1 *time units.*

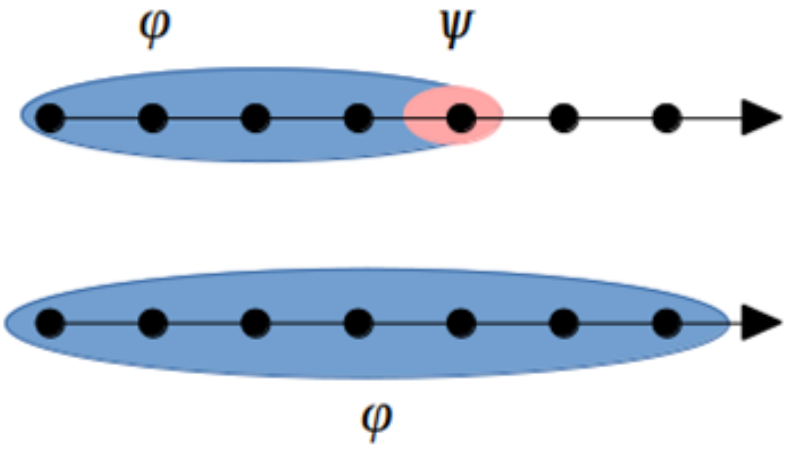

Figure 2.7: Representation of the Release LTL operator.

Source: [Losada, 2020]

- R_1: *It is always the case that if '*$(STATE == STATE\_Warmup)$ *and* $(CurrentTemp < DesiredTemp)$*' holds, then* $(STATE == STATE\_Warmup)$ *and* $(ShowWarmupTemp == true)$ *holds for at least* 100 *time units.*

Both R_0 *and* R_1 *can be formalized in LTL temporal logic, with these formulae:*

- $R_0 \equiv \varphi_0 = G(STATE == Start\_Heater \implies \mathcal{X}(STATE == Warmup))$
- $R_1 \equiv \varphi_1 = G((STATE == STATE\_Warmup) \wedge (CurrentTemp < DesiredTemp) \implies$
  $\mathcal{X}(\neg(STATE == STATE\_Warmup) \wedge (ShowWarmupTemp))) \wedge$
  $\mathcal{X}\mathcal{X}(\neg(STATE == STATE\_Warmup) \wedge (ShowWarmupTemp))) \wedge$
  $\ldots$
  $\wedge\, \mathcal{X}^{100}(\neg(STATE == STATE\_Warmup) \wedge (ShowWarmupTemp)))$

Then, if the specification wanted the conjunction of both requirements, it the would be:
$\varphi = \varphi_0 \wedge \varphi_1$

Some properties of LTL

We will mention some essential properties of LTL.

To begin with, distributivity holds for some operators. For instance, it does not hold for G and $\vee$, but it does for G and $\wedge$. Therefore, for instance this $\mathcal{X}(\phi \vee \psi) \equiv (\mathcal{X}\phi) \vee (\mathcal{X}\psi)$ and this $(\phi \wedge \psi)\,\mathcal{U}\; p \equiv (\phi\,\mathcal{U}\; p) \wedge (\psi\,\mathcal{U}\; p)$ hold.

As for the negation propagation, we can see (1) X is self-dual, (2) F and G are dual, (3) U and R are dual. Thus, for instance, this $\neg\mathcal{X}\phi \equiv \mathcal{X}\neg\phi$, this $\neg F\phi \equiv G\neg\phi$ and this $\neg(\phi\;\mathcal{R}\;\psi) \equiv (\neg\phi\;\mathcal{U}\;\neg\psi)$ hold.

Also, LTL holds some own special properties related to temporality. Some are listed below:

- $F\phi \equiv FF\phi$
- $G\phi \equiv GG\phi$
- $\phi\;\mathcal{U}\;\psi \equiv \phi\;\mathcal{U}\;(\phi\;\mathcal{U}\;\psi)$
- $\phi\;\mathcal{W}\;\psi \equiv \psi \vee (\phi \wedge \mathcal{X}(\phi\;\mathcal{W}\;\psi))$
- $G\phi \equiv \phi \wedge \mathcal{X}(G\phi)$

Note that there are more similar properties that have not been included to avoid redundancy.

Also, note that CTL (see Subsubsection 2.1.2) and LTL are both a subset of CTL* [8], but are incomparable. For example:

- No formula in CTL can define the language that is defined by the LTL formula $F(Gp)$.
- No formula in LTL can define the language that is defined by the CTL formula $AG(p \implies (EXq \wedge EX\neg q))$ nor the formula $AG(EF(p))$.

However, a subset of CTL* exists that is a strict superset of both CTL and LTL.

Past LTL

Analogously to what has been explained in Subsubsection 2.1.3 above, we introduce Past LTL: LTL with modalities that refer to the past.

[8] https://en.wikipedia.org/wiki/CTL*

LTL provides past operators: $\overline{\Box}$, $\overline{\Diamond}$, $\overline{\circ}$, $\widehat{\overline{\circ}}$, $\mathcal{S}$ and $\mathcal{B}$

Their informal semantics is as follows:

- $\overline{\Box}$ is used to express a property that holds on all previous states.
- $\overline{\Diamond}$ is used to express that a property must hold in some previous state.
- $\overline{\circ}$ or $\mathcal{Z}p$ or *Zyesterday* is used to express a property that must hold in the previous state and there must be one such previous state.
- $\widehat{\overline{\circ}}$ or $\mathcal{Y}$ or *Yesterday* is used to express a property that must hold at the previous state if it exists, otherwise it is true.
- $p\ \mathcal{S}\ q$ is used to express that after q holds then p must hold in all subsequent states.
- $p\ \mathcal{B}\ q$ is very similar to the previous one but allows q never holding, in this case, making p mandatory to hold in all previous states.

Note that the two Yesterday forms ($\mathcal{Z}$ and $\mathcal{Y}$) really are not the same: indeed second one is *weaker* version (defined like it by [Tonetta, 2017]) of the first one that is true in the initial state. Formally, $\mathcal{Z}(\bot) = \bot$, while $\mathcal{Y}(\bot) = \top$.

Also note that some properties change between both operators. We can see an example of it in Lemma 2.2 below.

Lemma **2.2.** *[Negation distribution property difference between yesterday operators]*

The negation distribution property is held by both $\mathcal{Z}$ and $\mathcal{Y}$: that is, $\neg\mathcal{Z}(\phi) \equiv \mathcal{Z}(\neg\phi)$ and $\neg\mathcal{Y}(\phi) \equiv \mathcal{Y}(\neg\phi)$, except when $\phi = \bot$.

Proof. We can check it easily by cases:

- In $\mathcal{Z}$: Since $\mathcal{Z}(\bot) = \bot$, then in the initial state, $\neg\mathcal{Z}(\bot) \equiv \mathcal{Z}(\neg\ \bot)$, since $\neg\ \bot = \mathcal{Z}(\neg\ \bot)$. Thus, it always the case that $\neg\mathcal{Z}(\phi) \equiv \mathcal{Z}(\neg\phi)$.
- In $\mathcal{Y}$: When $\phi = \bot$, then, since $\mathcal{Y}(\bot) = True$, $\neg\mathcal{Y}(\bot) \equiv \mathcal{Y}(\neg\ \bot)$ because $\neg\mathcal{Y}(\bot) = \bot$ and $\mathcal{Y}(\neg\ \bot) = True$.

Then, both operators behave the same way with respect to the negation except in the initial step.

□

These $\mathcal{Z}(\bot)$ and $\mathcal{Y}(\bot)$ situations are called *falling off the trace* (see Definition 3.14).

Apart from this, as for the expresiveness of Past LTL, it is known that standard LTL when extended with past-time modalities does not have more expressive power than LTL with only future modalities, but some properties can be defined in an exponentially more succinct manner [Markey, 2003].

2.1.4. Reactive Systems: Realizability vs Satisbiability

Depending on the type of systems, satisfiability is not a sufficiently expressive concept for industrial applications. This can be better understood in the subsubsections below.

Closed systems and open systems

Computer science is nowadays not only working with (1) programs that transform data and then terminate, but also with (2) non-terminating systems that have to interact with a (possibly evil) environment. An example of this is the controller (i.e. system) that checks the anti-lock braking system in a car: it receives a constant input stream of sensor readings, such as the wheel speed of each wheel, and using that information, it selectively applies the brakes on one wheel to maintain a uniform wheel speed.

We can, then distinguish the following concepts: *closed systems* and *open systems*:

1. Closed systems: Its behavior is completely determined by the state of the system once it has received the initial input.
2. Open systems (i.e. reactive systems): There is an interaction (a constant flow of input and reaction) between the system and its (maybe antagonistic) environment, so the combined behavior depends on this interaction.

An example of each can be given based on a drink dispensing machine:

- In a closed system, the environment can not modify any of the system (i.e. controllable) variables:
    1. The machine cyclically boils water.
    2. The machine makes an nondeterministic choice without consulting the environment.

3. The machine serves either coffee or tea

- In an open system, the environment can modify some of the system variables (generally, some input ones):

1. The machine cyclically boils water.
2. The environment to chooses between coffee and tea (i.e. a system variable is modified).
3. The machine deterministically serves a drink according to the environment's choice.

The reason to make the difference between these two system types is that, as said in the introduction of this Subsection, temporal logic (see Subsubsection 2.1.2) specifications which are satisfiable at the system level may be inconsistent (i.e. unrealizable) when interpreted over an open (i.e. reactive) system.

Consistency in closed systems

The *satisfiability* concept can only be used if we are talking about consistency in closed systems, not open ones. We can describe satisfiability in terms of systems in Definition 2.1 below.

***Definition* 2.1.** *[Satisfiability in terms of systems] A formula φ is said to be satisfiable if and only if there exists a closed system $\mathcal{S}$ that satisfies φ.*

Formally, taking (1) a function SAT that evaluates if a formula is satisfiable, and (2) an arbitrary formula $\varphi(\overline{v})$, where $\overline{v}$ is the set of system variables then:

$$SAT(\varphi(\overline{v})) \leftrightarrow (\exists \overline{v} :: \varphi(\overline{v}))$$

In LTL this has to hold in a trace.

Note that it is a nontrivial task to find whether a specification is satisfiable, whereas their satisfiability checking complexities are (1) PSPACE-complete for LTL and (2) EXPTIME-complete for CTL.

Also, note that two or more satisfiable formulas may together result in an unsatisfiable specification. For instance, without need of temporality, we can see that $\varphi_1 \equiv a \implies b$ is satisfiable, and so is $\varphi_2 \equiv a \implies \neg b$; but not their conjunction $\varphi \equiv \varphi_1 \wedge \varphi_2$.

Using the terminology of games (see Section 2.2), we could say it is a game of a single player that owns all the variables in the system. Therefore, this single player is called the *system*.

Consistency in reactive systems

Open systems can be characterized between them depending on the power they give to the environment. That power can be measured depending to the quantification that the variable of each variable of the player is operated with: from existential to universal. This can be seen in Figure 2.8 below, where variables v_1 to v_n belong to a single player.

Figure 2.8: Different power of a player depending on the quantification of its v_1 to v_n variables.

Source: Own

Note that the system will always have its variables existentially quantified, since it makes no sense to it to make its own possible reactions more restrictive.

Therefore, the standard in the industry is that the for all the choices of the environment, the system must have at least one choice to react: that is, the system has to find an existential combination of its variables, against a universal combination of the variables of the environment. This can be seen in the 2-dimension Figure 2.9.

In short, an open system has to be correct with respect to any environment, especially when the controllable part of the system cannot collaborate with the environment (i.e. when the environment is expected to be evil or unpredictable).

Therefore, a new question arises: the issue of *implementability*, commonly known as *realizability* [Pnueli and Rosner, 1989], which is described in Definition 2.2 below.

***Definition* 2.2.** *[Realizability] A formula φ is said to be realizable if and only if (1) there exists a module M which satisfies φ under any environment and (2) M is not clairvoyant (i.e. a nondeterministic guesser).*

Note that the variables are splitted between the variables of the environment and the variables of the system.

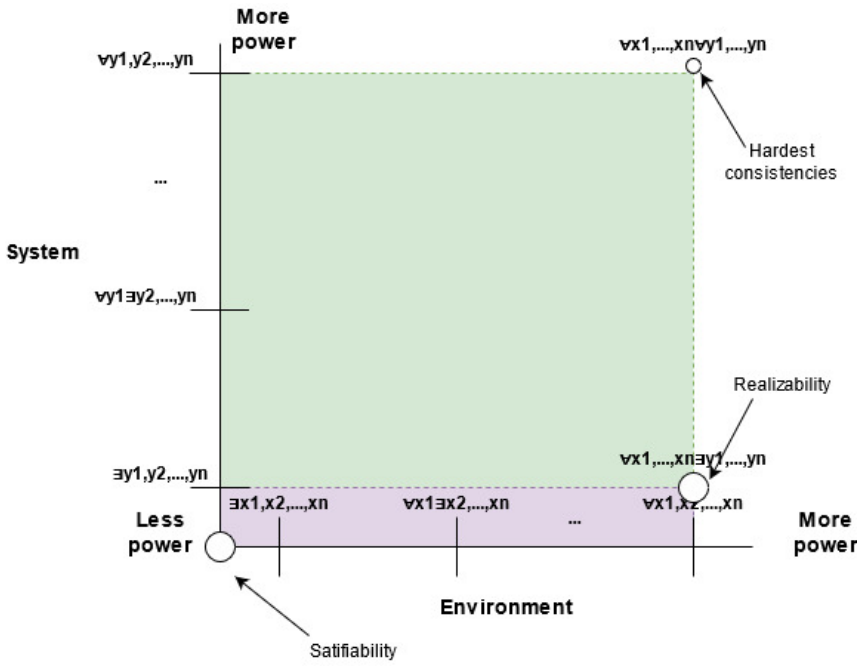

Figure 2.9: Different power paradigms in a two-player system: the purple colour area indicates the realizability (see Definition 2.2), while the green one shows different levels of system´s power; in addition, the satisfiability (see Definition 2.1), the realizability and the *hardest consistencies'* (i.e. all the variables are universally quantified) points are indicated.

Source: Own

In some cases, satisfiability and realizability can be the same (when the environment cannot have any power).

But, generally, the specifications that have been designed separating the system and the environment give certain power to the environment and, indeed, the designer has to consider all the possibilities of the environment, that is, a completely evil environment.

Therefore, if a specification is satisfiable, it may not be realizable. In addition, obviously, if the specifications are not satisfiable (i.e. cannot reach a satisfying position not even with both players' collaboration), then they are not realizable either.

In summary, in this thesis we will be moving in this expressivity dimension: linear temporal reactive specifications, which can be graphically seen in Figure 2.10 below.

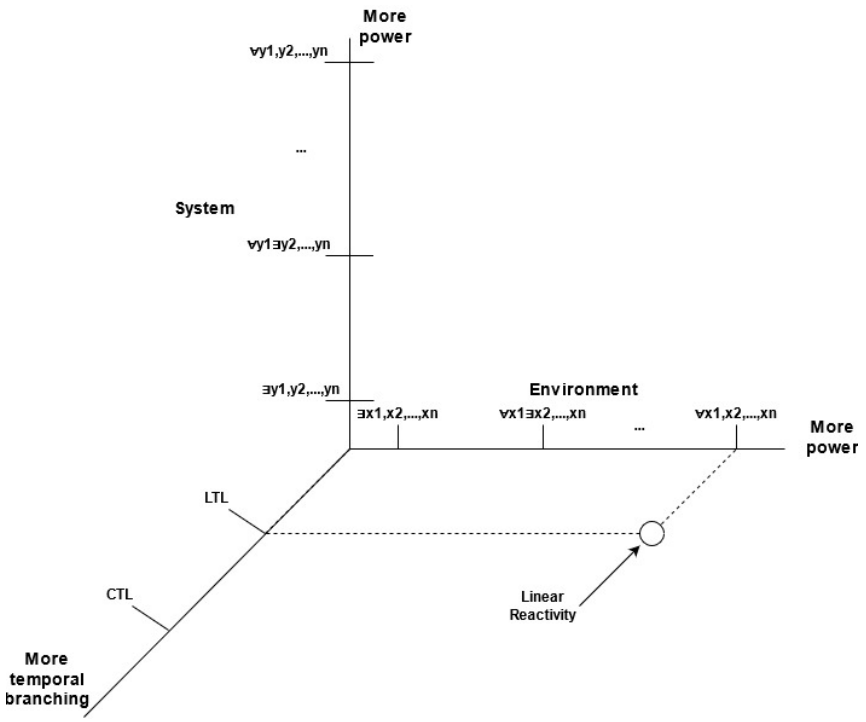

Figure 2.10: Expressivity of the specification that we are generally talking about: LTL + reactivity.

Source: Own

2.2. Infinite games, Realizability and Synthesis

Over the last years, it turned out to be very fruitful to model and analyze reactive systems in a game-theoretic framework, which captures the antagonistic and strategic nature of the interaction between the system and its environment.

More formally, this solution can be traced back to the synthesis problem for Boolean circuits, nowadays known as Church's problem (see Definition 2.3 below).

***Definition* 2.3.** *Given a specification $\mathcal{S}$ on the input-output behavior of circuits expressed in some suitable formalism, find a circuit that satisfies the given requirement (or determine that there is no such circuit).*

So it turns out Church's problem can be interpreted as a game between two agents: an environment generating an (evil) infinite stream of input bits, each of which is answered by an output bit generated by the circuit (i.e. the system or the controller). The requirement on the input-output behavior determines the winner of each execution: if the pair

of bitstreams satisfies the requirement, then the circuit wins, otherwise the environment wins.

In this view, Church's problem boils down to finding a finitely represented rule which prescribes for every finite sequence of input bits an output bit such that every input stream is answered by an output stream in a way that the pair of streams satisfies the given requirement. This is called an *strategy* (see Definition 2.7).

2.2.1. Games, Arenas, Plays and Strategies

We will define a set of fundamental concepts, as follows: arenas (see Definition 2.4), sub-arenas (see Definition 2.5), plays (see Definition 2.6), games (see Definition 2.10, strategies (see Definition 2.7), consistency (see Definition 2.8), positional strategies (see Definition 2.9), winning strategies (see Definition 2.11) and winning regions(see Definition 2.12).

Arenas and plays

The idea of an arena is to describe the rules the two players have to follow and additionally, the order in which the players make their moves.

To get an intuition of what an arena looks like, consider the graphical example in Figure 2.11 below, where the round vertices describe the vertices owned by Player 0, the angled ones the vertices owned by Player 1. The edges describe the moves the players can do.

The concept of the game is formally described in Definition 2.4 below.

***Definition* 2.4.** *[Arenas] An arena $\mathcal{A} = (V, V_0, V_1, E)$ is a tuple where:*

- *V is a finite set of vertices, some of them are initial.*
- *$V_0 \subseteq V$ is the set of vertices owned by Player 0.*
- *$V_1 = V \backslash V_0$ is the set of vertices owned by Player 1.*
- *$E \subseteq V \times V$ is a set of directed edges.*

We could also want to take some part of an arena, producing a **sub-arena** (see Definition 2.5 below), like the one in Figure 2.12.

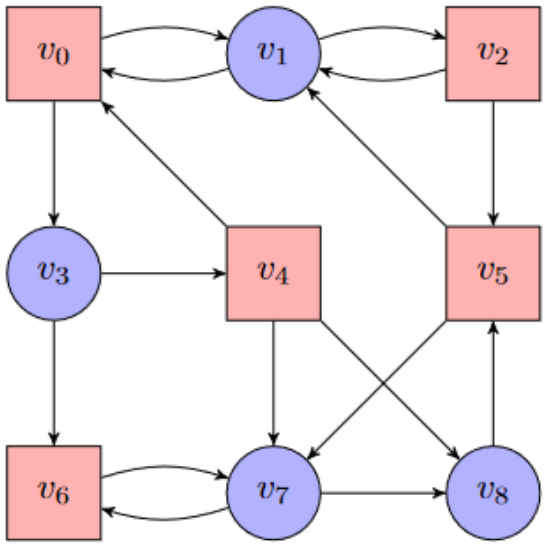

Figure 2.11: Graphical example for an arena.

Source: [Zimmermann and Klein, 2014]

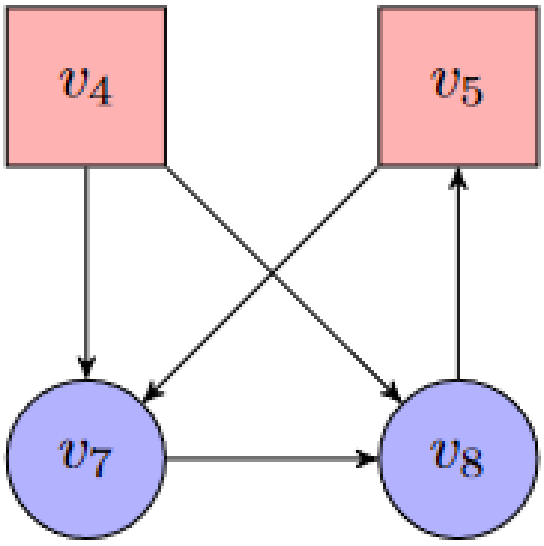

Figure 2.12: A sub-arena of the arena in Figure 2.11

Source: [Zimmermann and Klein, 2014]

***Definition* 2.5.** *[Sub-arenas] Let $\mathcal{A} = (V, V_0, V_1, E)$ be an arena and $V' \subseteq V$ such that every vertex in V' has a successor vertex in V'. The sub-arena of A, denoted by sub$(\mathcal{A})$, is defined as:*

$$sub(V') = (V \cap V', V_0 \cap V', V_1 \cap V', E \cap (V' \times V'))$$

Players move through the arena by making decisions. If they do that forever, those moves

define an infinite sequence of vertices they visits. Such an infinite sequence we also call a *play* (see Definition 2.6 below) on the arena. An example for a play is given in Figure 2.13 below.

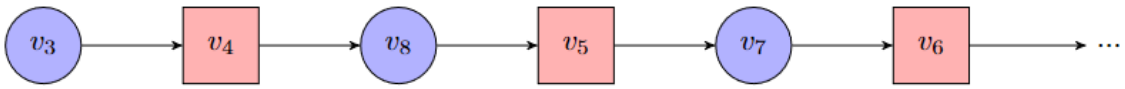

Figure 2.13: A play over the arena in Figure 2.11

Source: [Zimmermann and Klein, 2014]

***Definition* 2.6.** *[Plays] A play on an arena $\mathcal{A}$ is an infinite sequence of positions $\rho = \rho_0, \rho_1, \ldots \in V^\omega$ such that for all $n \in \mathbb{N}$ $(\rho_n, \rho_{n+1}) \in \mathcal{T}$.*

Plays, strategies and games

We say that a strategy (see Definition 2.7 below) may depend on the history, where the history contains the finite prefix of a play that has already been played.

To describe this formally, we define a strategy of a player as a function from the actual history of moves into the next outgoing edge. This way we fix a possible behaviour of that player without considering the strategy of the other player.

***Definition* 2.7.** *[Strategies] A strategy for a Player i in an arena $\mathcal{A}$ is a function $\sigma : \mathcal{V} * \mathcal{V}_i \to \mathcal{V}$ such that whenever $\sigma(w, v) = v'$, then it holds that: $(v, v') \in \mathcal{T}$. That is, taking a history and the current position, player i chooses the move to the successor position.*

If a play then results from using this strategy, we call it consistent with the strategy (see Definition 2.8 below).

***Definition* 2.8.** *[Consistency] A play ρ on an arena $\mathcal{A}$ is consistent with a strategy σ in $\mathcal{A}$ iff for all $n \in \mathcal{N}$ with $\rho_n \in V_i$ we have that $\sigma(\rho[n]) = \rho_{[n+1]}$. We denote the set of all plays consistent with σ and starting in some vertex $v \in \mathcal{V}$ with $Plays(\mathcal{A}, \sigma, v)$.*

Note that if a play then results from using a strategy, we call it *consistent with the strategy*.

One might argue that Definition 2.8 presents a too expressive concept of *strategy*; too expressive for practical usage, since it may have to fix infinitely many different decisions.

Thus, we present a weaker notion of strategy: *positional strategies* (see Definition 2.9 below), where the strategy is not allowed to depend on the history. This means at each

vertex the strategy chooses always the same successor and, consequently, with a positional strategy, a player has always to chose the same outgoing edge if the token is in a vertex he owns.

***Definition* 2.9** (Positional strategies)**.** *A strategy σ for Player i in an arena $\mathcal{A}$ is positional iff $\sigma(wv) = \sigma(v)$ for all $w \in V^*$ and $v \in V_i$.*

We will introduce the concept of a *game* (see Definition 2.10 below) by introducing some kind of *winning conditions*. So far, the players are only able to play infinitely long on an arena, but there is no notion of when a player is *winning* that play: so we will take the set of all possible plays and specify a subset of this set and say that, for instance, Player 0 is winning the play if it is in the set.

***Definition* 2.10.** *[Games] A game $\mathcal{G} = (\mathcal{A}, Win)$ is a tuple containing an arena $\mathcal{A}$ and a set of winning plays $Win \subseteq Plays(\mathcal{A})$. We call a play ρ winning for a Player 0 if and only if $\rho \in Win$ and winning for another Player 1 otherwise.*

Note that, since we are in a safety game (see Subsubsection 2.2.2), the objective for one player will be to remain on the winning set, while the other one will try to win the dual reachability game (see Subsubsection 2.2.2). We will introduce these two concepts below.

To finish, we can now introduce the notion of winning strategies (see Definition 2.11 below) per initial position.

***Definition* 2.11.** *[Winning strategies] Let $\mathcal{G} = (\mathcal{A}, Win)$ be a game with and σ be a strategy for Player i on $\mathcal{A}$. The strategy σ is a winning strategy from vertex $v \in \mathcal{V}$ for Player i iff every play $\rho \in Plays(\mathcal{A}, v)$ consistent with σ is winning for Player i.*

And, from that, the the notion of a winning region (see Definition 2.12 below).

***Definition* 2.12.** *[Winning regions] The winning region $\mathcal{W}_i(\mathcal{G})$ of a game $\mathcal{G}$ is defined for Player i as the set of vertices $v \in \mathcal{V}$ for which there exists a winning strategy starting from vertex v for Player i.*

Note that, for all games $\mathcal{G}$ *determinacy* holds [9]: that is, $\mathcal{W}_0(\mathcal{G}) \cap \mathcal{W}_1(\mathcal{G}) = \varnothing$.

Using these definitions, we can construct any complex infinite game such as *Büchi*, *Parity* or *Müller Games*. But those game types are out of the scope of this thesis, and we are interested in the above mentioned safety and reachability games.

[9] https://en.wikipedia.org/wiki/Determinacy

2.2.2. Reachability Games and Safety Games

In this thesis, we are interested in specifications that are written to guarantee safety properties (see Subsubsection 2.2.2 below). As a result, we will only be using reachability (see Subsubsection 2.2.2) and safety games (see Subsection 2.2.2).

Safety properties

Bur first of all, what are safety properties? They express that certain *bad* situations never happen. This contrasts with *liveness properties*, which stipulate that certain *good* situations must eventually happen.

Synthesis algorithms for safety specifications can be useful even for specifications that contain liveness [10] properties (i.e. *something good will eventually occur*). One example is bounded synthesis, whose approaches can reduce the synthesis of more sophisticated specifications, such as those expressed in LTL, to safety problems problems by setting a limit on the reaction time. For example, instead of requiring that some event happens eventually, one may require that it occur within the first k steps. Clearly, a realization (see Definition 2.2) of the latter is also a realization of the former. By choosing k as low as possible (so that so that a solution still exists), we can even obtain systems that react faster.

Note that these properties can be refuted with a finite trace (using QBF solvers [11], for instance), in which the environment assigns values to its variables reaching a state in which, whatever the state in which, whatever the system does, it is not achievable.

It is also worth adding that one of the parallel lines of the thesis included finding these traces. This process can be improved by using as a time trace generator the [12] application from [Arteche and van der Hallen, 2020].

Reachability games

If a safety property (see Subsubsection 2.2.2 above) consists in avoiding certain event, a reachability property is the proper dual: to reach a certain event.

[10] https://en.wikipedia.org/wiki/Liveness
[11] http://www.florianlonsing.com/talks/Lonsing-SAT-SMT-school-india-2017.pdf
[12] https://github.com/alephnoell/QBDef

In terms of games, we are interested whether Player 0 is able to move the into a specific area of the arena, where an area is just a set of vertices. We call this winning condition a reachability condition, whereas the set of vertices, from which at least one vertex should be reached, is called the reachability set.

A game with a reachability condition as a winning condition is a reachability game (see Definition 2.13 below).

***Definition* 2.13.** *[Reachability games] Let the $\mathcal{O}(\rho)$ mean that a ρ play occurs infinitely many often, and let reachability condition $reach(R)$ on a set $R \subseteq V$ for an arena $A = (V, V_0, V_1, E)$ be defined as:*

$$reach(R) := \{\rho \in Plays(\mathcal{A}) | \mathcal{O}(\rho) \cap R = \varnothing\}$$

Then we call the game $\mathcal{G} = (\mathcal{A}, reach(R))$ a reachability game with reachability set R. In other words, a play $\rho \in Plays(\mathcal{A})$ is in $reach(R)$ if for some n $p_n \in R$.

An example for a reachability game is given in Figure 2.14 below, where we graphically denote the reachability set by using doubly framed vertices.

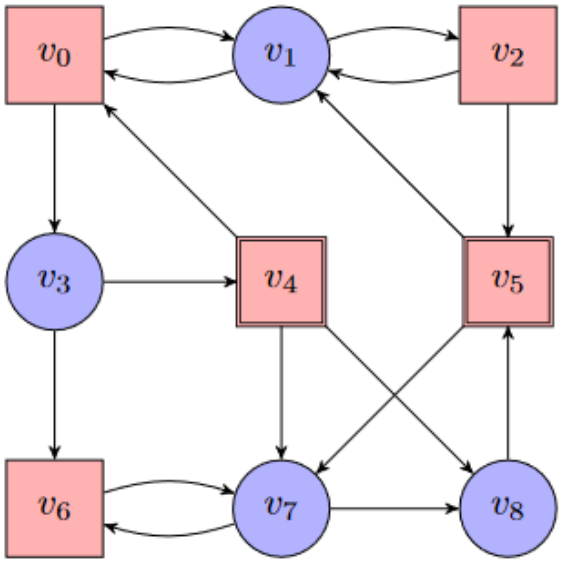

Figure 2.14: An arena of a reachability game.

Source: [Zimmermann and Klein, 2014]

In Figure 2.14 above we can see that, for example if the system starts in a vertex of the reachability set, like in vertex v_5, he already have won so it does not matter which

successor he or his opponent chooses. He also win if he starts in a vertex next to the reachability set that he controls, like vertex v_8.

The problem is what to do in the rest of the cases that cannot reach the reachability set in one step. This is solved using a recursive construction called *attractors*, whose study is out of the scope of this thesis.

Safety games

As said, dual to the reachability games, we have the safety games, whose winning condition is the so called *safety condition*.

Where for the reachability condition Player 0 is asked to reach a specific region of the arena, in the safety condition Player 0 is not allowed to leave a specific region. We also call this region the safe region of the arena.

A game with a safety condition as a winning condition is a safety game (see Definition 2.14 below).

***Definition* 2.14.** *[Safety games] Let the $\mathcal{O}(\rho)$ mean that a ρ play occurs infinitely many often, and let safety condition safe(S) on a set $S \subseteq V$ for an arena $A = (V, V_0, V_1, E)$ be defined as:*

$$safe(S) := \{\rho \in Plays(\mathcal{A}) | \mathcal{O}(\rho) \subset S\}$$

Then we call the game $\mathcal{G} = (\mathcal{A}, safe(S))$ a safety game with safety set S.

An example for a safety game is given in Figure 2.15 below, where we graphically denote the safety set by using doubly framed vertices.

In Figure 2.15 above we can see that, Player 0 can win exactly from the set $\mathcal{W}_0 = v_1, v_2, v_5, v_7, v_8$ since in v_4 Player 1 can move to the unsafe vertex v_0 and from v_3 Player 0 can only move to v_6, which is unsafe, or v_4 from which we already have seen that Player 1 can move to an unsafe region.

Note that in the game above, the goal of both players exactly have swapped in comparison to reachability games. Player 1 now tries to reach the unsafe region of the arena and Player 0 has to avoid this.

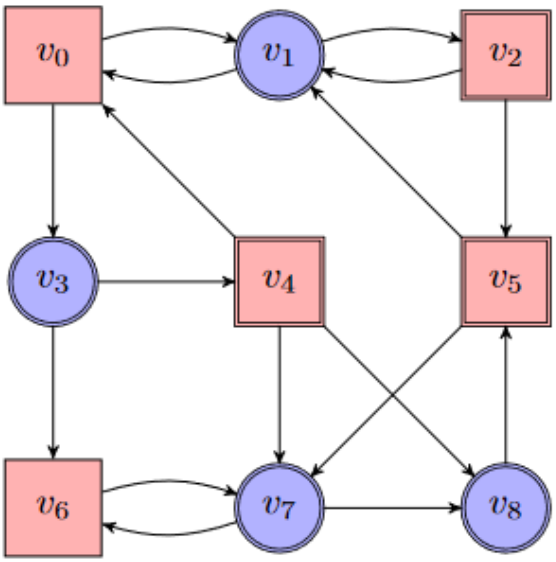

Figure 2.15: An arena of a safety game.

Source: [Zimmermann and Klein, 2014]

2.3. Decision Procedures: Quantifier Elimination

In order to determine whether a set of requirements written in a given first order theory is satisfiable/realizable (see Subsection 2.1.4) or not, the standard method is to use a suitable *decision procedure* (see Description 2.9 below) procedure designed for the given theory.

Description **2.9.** *A decision procedure is an algorithm that, given a decision problem, terminates with a correct yes/no answer.*

In practice, in order to solve formulae written in a given first order theory, the standard is to use an SMT solver [13]; or a QBF solver [14] in case there are only Boolean variables in the formula.

In any case, both techniques internally make use of one particular decision procedure: *quantifier elimination* (see the immediately next Subsection 2.3.1).

[13] https://en.wikipedia.org/wiki/Satisfiability_modulo_theories
[14] https://www.cs.utexas.edu/users/hunt/class/2015-fall/cs395t/slides/QBF.pdf

2.3.1. Basics on quantifier elimination

We will now present the concrete decision procedures in which we are interested for this thesis: quantifier elimination [15], which is interesting for this thesis not only because of their main production per se, but because some modifications of it can lead us to interesting results: that is, we rely on quantifier elimination for some Booleanization algorithms (see Section 4.2).

Main idea of quantifier elimination

A quantifier elimination procedure (QEP) or method is a sound [16] and complete [17] algorithm that constructs from a given formula an equivalent quantifier-free formula: that is, it eliminates quantifiers from a formula to produce an equivalent quantifier-free formula .

When the given formula does not have any free variables (i.e. it is a sentence [18]), the atoms of the resulting formula are applications of predicates to constant terms such as $(3 < 5)$. If the truth value of these constant terms is decidable, a quantifier elimination procedure provides a basis for a satisfiability decidable procedure: that is, if ground atoms are decidable, then quantifier elimination yields a decision procedure.

When the given formula contains free variables, the resulting quantifier-free formula contains a subset of these free variables.

We will first present the method of quantifier elimination in an abstract context, although we are focusing in some concrete QEP for the theory of integers $\mathcal{T}_{\mathbb{Z}}$ (i.e. *Cooper's method* in [Cooper, 1972]) and the theory of rationals $\mathcal{T}_{\mathbb{Q}}$ (i.e. *Ferrante and Rackoff's method* in [Ferrante and Rackoff, 1975]), respectively.

The complexity of these algorithms are near-optimal in time complexity, although quantifier-elimination methods are, in general, computationally expensive.

Mechanical idea of quantifier elimination

The idea is to eliminate quantifiers of a formula F until only a quantifier-free formula G that is equivalent to F remains.

[15] https://en.wikipedia.org/wiki/Quantifier_elimination
[16] https://en.wikipedia.org/wiki/Soundness
[17] https://www-users.cse.umn.edu/~gini/4511/search
[18] https://en.wikipedia.org/wiki/Sentence_(mathematical_logic)

Formally, a theory $\mathcal{T}$ admits quantifier elimination if and only if there is an algorithm that, given a $\Sigma_{\mathcal{T}}$-formula F, returns a quantifier-free $\Sigma_{\mathcal{T}}$-formula G that is $\mathcal{T}$-equivalent to F; where $\mathcal{T}$ is decidable if and only if satisfiability in the quantifier-free fragment of $\mathcal{T}$ is decidable.

If a formula F contains free variables, then a quantifier elimination procedure produces an equivalent quantifier-free formula F′ such that free(F′) ⊆ free(F).

A key remark: Universally quantified to existentially quantified

We need only consider formulae of the form $\exists x.\ F$ for quantifier-free formula F . For given arbitrary formula G, choose the innermost quantified formula $\exists x.\ G$ or $\forall x.\ G$.

In the latter case, we can rewrtie ∀x. G as ¬(∃x. ¬G) and focus on the subformula ∃x. ¬G inside the negation. We can see a use case in Example 2.4 below.

Example **2.4.** *[Universal to existential]*

Let us consider the arbitrary 3-variable $\Sigma_{\mathcal{T}}$-formula $\phi_0 : \forall x.\ \forall y.\ \exists z.\ F_0[x,y,z]$, where the theory $\mathcal{T}$ admits quantifier elimination.

Now, assume we apply QE to the innermost formula $\exists z.\ F_0[x,y,z]$, obtaining the next result:

$$\phi_1 : \forall x.\ \forall y.\ F_2[x,y]$$

Instead of getting stuck in the QE procedure because of the universal quantifier, we can rewrite the innermost formula $\forall y.\ F_2[x,y]$ as follows:

$$\neg(\exists y.\ \neg F_2[x,y])$$

In the context of this thesis, this remark is relevant, since we will use universal quantifiers for the environment's variables.

Relation of Quantifier Elimination with decidability

In early model theory [19], quantifier elimination was used to demonstrate that various theories possess properties like decidability and completeness. A common technique was to

[19] https://en.wikipedia.org/wiki/Model_theory

show first that a theory admits elimination of quantifiers and thereafter prove decidability or completeness by considering only the quantifier-free formulas. This technique can be used to show that Presburger arithmetic is decidable.

Theories could be decidable, yet not admit quantifier elimination. Strictly speaking, the theory of the additive natural numbers did not admit quantifier elimination, but it was an expansion of the additive natural numbers that was shown to be decidable. Whenever a theory is decidable, and the language of its valid formulae is countable, it is possible to extend the theory with countably many relations to have quantifier elimination (for example, one can introduce, for each formula of the theory, a relation symbol that relates the free variables of the formula).

We will now present (see the immediately next Subsection 2.3.2) the first QEP; which, indeed was not created with the intention to be a QEP, but it serves as a foundation for the rest of them.

2.3.2. The Fourier-Motzkin algorithm

This method is considered to be the first quantifier elimination method ever raised. It will be helpful as (1) an example of an algorithmic method do do quantifier elimination and (2) it is the foundation of test-point based quantifier eliminations we will see in Subsection 2.3.3.

Description **2.10.** *Fourier–Motzkin elimination, also known as the FME method, is a mathematical algorithm for eliminating variables from a system of linear inequalities.*

Note that the method can output both integer and real solutions, depending on the used theory.

The algorithm is named after Joseph Fourier and Theodore Motzkin who independently discovered the method in 1827 and in 1936, respectively.

The FME method

There are different versions of the FME method, but the essence is the same: the elimination of a set of variables $\mathcal{V}$, from a system of relations (i.e. linear inequalities) is a the creation of another system of the same sort, but without the variables in $\mathcal{V}$, such that both systems have the same solutions over the remaining variables.

If all variables are eliminated from a system of linear inequalities, then one obtains a system of constant inequalities (i.e. numeric predicates). It is then trivial to decide whether the resulting system is true or false. It is true if and only if the original system has solutions. As a consequence, elimination of all variables can be used to detect whether a system of inequalities has solutions or not: that is, as a decision procedure for numeric formulae.

We can see the result of applying one iteration of the FME method in Example 2.5 below.

Example **2.5.** *Let us consider the following system of inequalities belonging to the theory of rationals:*

$$\begin{aligned} 2x - 5y + 4z &\leqslant 10 \\ 3x - 6y + 3z &\leqslant 9 \\ -x + 5y - 2z &\leqslant -7 \\ -3x + 2y + 6z &\leqslant 12 \end{aligned}$$

To eliminate x, we can write the inequalities in terms of x:

$$\begin{aligned} x &\leqslant \frac{10 + 5y - 4z}{2} \\ x &\leqslant \frac{9 + 6y - 3z}{3} \\ x &\geqslant 7 + 5y - 2z \\ x &\geqslant \frac{-12 + 2y + 6z}{3} \end{aligned}$$

We now have two inequalities with leq and two with $\geqslant$*; the system has a solution if and only if the right-hand side of each leq inequality is at least the right-hand side of each* $\geqslant$ *inequality. We have* $2 * 2$ *such combinations:*

$$\begin{aligned} 7 + 5y - 2z &\leqslant \frac{10 + 5y - 4z}{2} \\ 7 + 5y - 2z &\leqslant \frac{9 + 6y - 3z}{3} \\ \frac{-12 + 2y + 6z}{3} &\leqslant \frac{10 + 5y - 4z}{2} \\ \frac{-12 + 2y + 6z}{3} &\leqslant \frac{9 + 6y - 3z}{3} \end{aligned}$$

We now have a new system of inequalities, with one fewer variable: that is, we have performed an existential variable elimination (this is better studied in Subsection 2.3.2).

Complexity of the FME method

We can see running an elimination step over n inequalities can result in at most $\frac{n^2}{4}$ inequalities in the output, thus running d successive steps can result in at most $4(\frac{n}{4})^{2^d}$ inequalities: that is, a double exponential complexity. This is due to the algorithm producing many unnecessary constraints (constraints that are implied by other constraints). The number of necessary constraints grows as a single exponential.

With FME we obtain a F' in DNF [20]. For a universal quantifier, through De Morgan's laws, we obtain a formula in CNF [21]. Such a naive algorithm suffers from an obvious inefficiency, particularly if applied recursively to formulas with alternating quantifiers.

For instance, consider $\exists x \forall y.\ F$. The algorithm will compute a CNF formula equivalent to $\forall y.\ F$, then convert this formula to DNF. Conversion from CNF to DNF through the application of distributivity of $\wedge$ over *vee* is extremely inefficient, even on propositional formulas. Furthermore, many conjunctions in the DNF are likely to be contradictory; that is, they will express incompatible linear constraints. It is therefore a waste of time and space to generate them. Finally, the DNF form obtained by distributivity may be needlessly complex; for instance, $(x < 0 \wedge x \geqslant 0) \wedge y > 0$ gets turned into $(x < 0 \wedge y > 0) \vee (x \geqslant 0 \wedge y > 0)$ whereas one should have merged both conjuncts into the more general $y > 0$.

So, FME elimination is a simple algorithm, yet, when it eliminates a single variable, the output conjunction can have a quadratic number of conjuncts compared to the input conjunction, thus a pass of simplification would be needed for practical efficiency.

For instance, unnecessary constraints may be detected using linear programming, but here we are considering the naive algorithm.

To sum up: in the one hand, (1) One FME-step for $\exists$ means that the formula size grows quadratically, in $O(2^n)$; whereas (2) a naive implementation of FME with m quantifiers of the form $\exists ... \exists$ has a cost of $O(m^{2^n})$. On the other hand, (3) with the concrete quantifier alternation $\exists \forall \exists \forall ... \exists$ a CNF/DNF conversion is required after each step, and, since this conversion is exponential, then the complexity is non-elementary.

[20] https://en.wikipedia.org/wiki/Disjunctive_normal_form
[21] https://en.wikipedia.org/wiki/Conjunctive_normal_form

Fourier-Motzkin is a Quantifier Elimination Procedure

Consider a system S of wide or strict linear inequalities, from which we wish to eliminate variable x. This means meaning that we wish to eliminate the quantifier from $\exists x.\ S$

Therefore, methods for eliminating variables from systems of linear inequalities do, in other words, eliminate an existential quantifier from a conjunction of linear inequalities. Thus, they are QEPs.

Thus, the FME method is also a QEP. We can check if the properties of a QEP hold over it:

- It preservers satisfiability:
  - The elimination of a set of variables, say V, from a system of relations (here linear inequalities) refers to the creation of another system of the same sort, but without the variables in V, such that both systems have the same solutions over the remaining variables.
  - This is the same that happens with QEP when comparing the $\mathcal{T}$-satisfiability of the first formula with the second one.
- The decision-procedure:
  - If all variables are eliminated from a system of linear inequalities, then one obtains a system of constant inequalities. It is then trivial to decide whether the resulting system is true or false. It is true if and only if the original system has solutions. As a consequence, elimination of all variables can be used to detect whether a system of inequalities has solutions or not.
  - This is the same that happens with QEP where if ground atoms are decidable, then QE yields a decision procedure.

Therefore, said by [Chaieb, 2006] we can conclude that the decidability of the theory of reals is arguably due to Fourier, even if it was later proved by Tarski.

Why Fourier-Motzkin is interesting: The foundation for the rest

We have seen lots of equivalences, such as $(\exists y.\ x < y \wedge y < z) \leftrightarrow (x < z)$ is an easy consequence of the axioms and the essence of FME [Nipkow, 2008].

However, one could think this QPE is not interesting, as the FME method requires DNF conversions, and the algorithm without a simplification step is, thus, non-elementary: the size of the system of inequalities can grow quadratically for each variable being eliminated, thus a complexity bound of $2^{2^{cn}}$, where n is the size of the original formula. The size of the coefficients of the inequalities can double [Monniaux, 2008].

Yet, this is not a relevant problem: the most popular QEPs are bounded by the same asymptotic complexity: $2^{2^{cn}}$. For instance, the *classical* algorithm for quantifier elimination over real or rational arithmetic is Ferrante and Rackoff's method [Ferrante and Rackoff, 1975], whose complexity is, again, $2^{2^{cn}}$. Note that Ferrante and Rackoff's algorithm never simplifies formulas, i.e. there is no CNF or DNF simplification on it.

On the other side, it has been stated that this algorithm only works for linear inequations, i.e. atoms of the form:

$$c + c_x x + c_y y + c_z z + ... \geqslant 0$$

Which construct formulae (or systems, if we make a conjunct) like:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

So FME cannot be expanded to non-linear theories: that is, we cannot use it to decide formulae with non-linear predicates.

One could enhance FME with a case discussion mechanism, but this enhancement would be limited to non-linear constraints where each variable appears in degree zero or one (otherwise an algorithm for solving semi-algebraic systems would need to support FME, which cannot really be considered as FME anymore). Moreover, this enhanced and parametric FME would no longer be able to rely on numerical methods for linear programming, thus losing a lot of practical efficiency like in [Monniaux, 2008]. Another approach like this is [Suriana, 2016].

Yet, the most typical QEPs for real arithmetics are (i.e. they performs quantifier elimination as FME, but over polynomial inequalities, not just linear), in short:

- Tarski-Seidenberg-Theorem: It states that a set in $(n+1)$-dimensional space defined

by polynomial equations and inequalities can be projected down onto n-dimensional space, and the resulting set is still definable in terms of polynomial identities and inequalities. It implies that quantifier elimination is possible over the reals.

- CAD: Although the original proof of the theorem was constructive, the resulting algorithm has a computational complexity that is too high for using the method on a computer. [Arnon et al., 1998] introduced the algorithm of cylindrical algebraic decomposition, which allows quantifier elimination over the reals in double exponential time. This complexity is optimal, as there are examples where the output has a double exponential number of connected components.

An important consequence of these QEPs is the decidability of the theory of real-closed fields (see [Monniaux, 2008]).

However, these techniques are not relevant for us. As it has been said in the beginning of the chapter, in this thesis we are not interested in quantifier elimination per se, but in the modification or use of QEPs to use them for other purposes: in our case, for the partition of an infinite game.

This is why we are interested in other kinds of QEPs, that allow us to do such partitions. Those QEPs work over Linear Arithmetics (in our case, Linear-Arithmetics-written requirements), so they do not take any idea of, for example, CAD, while they do rely on FME.

For instance, recently mentioned Ferrante and Rackoff's method can be understood as an extension of Fourier-Motzkin's algorithm to formulae with disjunctions [Monniaux, 2008]: in addition to checks for unbounded intervals, one looks at all couples, defining intervals, and checks that the middle point (or, for the matter, any point of the inside) verifies the formula.

Those QEPs, namely the test-point based QEPs, are the ones that are introduced in Subsection 2.3.3 below.

2.3.3. Test-points and quantifier elimination

A *test point method* involves identifying important intervals, and then *testing* a number from each interval.

Similarly, some QEPs make use of intervals to discretize the formula, so we could consider them test-point based QEPs.

The idea of the test-point method

Linear inequalities, like $\frac{3x}{2} - 1 \geqslant 5 - 7x$, can be solved by simply getting x all by itself on one side, and a number on the other side. The only thing different from working with equations is that if you multiply or divide an inequality by a negative number, then you must change the direction of the inequality symbol.

Nonlinear inequalities, like $x^2 < 3$ have the variable appearing with a certain multiplicity (in this case, 2), thus, they require more advanced tools, such as the test-point method (this method can also be used for linear inequalities, but one would be working much harder than needed to accomplish the task).

The idea is that every inequality can be written in a form with zero on the right hand side. The solution sets (i.e. the values of x that make each sentence true) are determined by which inequality symbol is used:

Test-point in Quantifier Elimination

In contrast to FME, test-point based algorithms do not perform DNF and CNF conversions and, instead they are NNF-based. As it has been stated in Subsubsection 2.3.2, if they added a step of transformation to CNF or DNF to their algorithm, then they would obtain a triple exponential: $2^{2^{2cn}}$.

There are some NNF-based algorithms based on the test point method (originally due to Cooper [Cooper, 1972], and Ferrante and Rackoff [Ferrante and Rackoff, 1975] and Weispfenning [Weispfenning, 1997]), whose idea is to find a finite set of test points T depending on ϕ such that:

$$\exists x.\ \varphi(x) = (\bigvee_{t \in \mathcal{T}} \varphi(t))$$

The complication is that $\mathcal{T}$ may contain values like infinity, infinitesimals or intermediate points, values that are not representable in the given term language.

For instance, Ferrante and Rackoff realized for linear real arithmetic that when eliminating x from φ it suffices to collect all lower bounds l of x (i.e. $l < x$ occurs in φ) and all upper bounds u of x (i.e. $x < u$ occurs in φ) and try all such $\frac{l+u}{2}$ as test points.

This idea is generally expressed in the Virtual Substitution algorithms (see [Sturm, 2017]).

3. CHAPTER

The Booleanization Theorem

This chapter describes the most important contribution of this thesis: the Temporal Booleanization theorem (see Subsection 3.1.1), together with a proof of this theorem (see Section 3.2) and a Booleanization algorithm based on it (see Subsection 3.1.2).

The Booleanization Theorem states that: (1) there is a correct Booleanization method for requirements and (2) if a numeric game is Booleanized, the realizability result of the resulting Booleanized game implies the realizability result of the original numeric game.

3.1. Wrong ideas, theorems and algorithm

The solution for the Booleanization problem requires first (1) an explanation of why this problem is not trivial (offered in Subsubsection 3.1.1), then (2) an overall argument that ensures that the solution is possible (offered in Subsection 3.1.1), (3) a concrete algorithm that performs what the theorem stated (offered in Subsection 3.1.2) and (4) a tempting method that is incorrect that justifies the presented one (see Subsection 3.1.3).

3.1.1. Final solution: A global Booleanization is possible

This subsection will in part, re-state the problem of Subection 1.1.2 and why this problem is not trivial. However, the main contribution is to deeply explain the solution of Subsection 1.2.2 and understand its methods and consequences.

The Booleanization problem in a nutshell

*This subsubsection will be a simplification of the explanation given in Subsection 1.2.1.

The realizability checking (see Subsubsection 2.1.4) of reactive numeric LTL requirements (see Subsubsection 2.1.3) is not possible by the state-of-the-art tools, since these tools handle only Boolean LTL formulae. Therefore, the requirements must be *Booleanized*.

Booleanizing a set of requirements consists of replacing all numeric literals (i.e. predicates) by Boolean variables.

When the requirements are reactive (i.e. model a two-player *environment vs system* game), this problem becomes particularly challenging.

The main issue is preserving the exact original power of each variable-owner of the numeric game (see an example in Subsubsection 3.1.3 below).

Our method in Section 3.1.1, proposes to give the ownership of all the Booleanized predicates to the one player and then managing the rest of the dependencies in a new requirement called the *extra requirement* (see Definition 3.10). This method is proven correct in Section 3.2.

Definitions for the Booleanization theorem

First, we introduce definitions and notation needed to understand both the Booleanization Theorem.

Variables. Short name: v_i

Throughout this thesis, *variables* do not refer to variables of an algorithm, but to variables in the purely mathematical sense. We can describe variables formally in Definition 3.1.

***Definition* 3.1.** *In requirements and formulae, a variable is a (potentially quantified) constituent symbol of a predicate (i.e. non-Boolean literal) or function. A variable can be numeric (i.e. integers, rationals...) or Boolean, and must belong to one of the variable-owners: whether the so-called environment or the so-called system.*

Bearing in mind Definition 3.1, we will usually refer to variables in a plural way: not considering just one of them, but the set of variables of each variable-owner. To do so, we will use the *overline* symbol (see Example 3.1 below).

Example **3.1.** *Let x be a numeric variable of the environment. We will denote this formally as follows:* $x \in \mathbb{E} \wedge x \in Num$*, where* $\mathbb{E}$ *denotes the set of the environment variables and Num denotes a numeric theory.*

Then, the set of all the numeric variables that belong to the environment is denoted like: $\overline{x} \in \mathbb{E} \wedge \overline{x} \in (2^{Num})$.

Concretely, we will use the following notation in our requirements (and games):

- $\overline{x}$: The numeric variables that belong to the environment. Formally:

$$\mathcal{V}^{\mathbb{E}}_{Num} = \{\overline{x} \in \mathbb{E} \mid \overline{x} \in (2^{Num})\}$$

- $\overline{y}$: The numeric variables that belong to the system. Formally:

$$\mathcal{V}^{\mathbb{S}}_{Num} = \{\overline{y} \in \mathbb{S} \mid \overline{y} \in (2^{Num})\}$$

  $\overline{y} \in \mathbb{S} \wedge \overline{y} \in (2^{Num})$

- $\overline{e}$: The Boolean variables that belong to the environment. Formally:

$$\mathcal{V}^{\mathbb{E}}_{\mathbb{B}} = \{\overline{e} \in \mathbb{E} \mid \overline{e} \in (2^{\mathbb{B}})\}$$

- $\overline{s}$: The Boolean variables that belong to the system. Formally:

$$\mathcal{V}^{\mathbb{S}}_{\mathbb{B}} = \{\overline{s} \in \mathbb{S} \mid \overline{s} \in (2^{\mathbb{B}})\}$$

Configurations. Short name: $conf_i$

Given a set of literals, a choice of each literal as positive or negative yields a configuration: that is, a configuration is a concrete combination on the positivity sign of the literals of the set of literals. It is formally described in Definition 3.2 below.

***Definition* 3.2.** *A configuration is a conjunction of literals, where each literal appears either in positive or in negative form.*

To illustrate this, see Example 3.2 below.

Example **3.2.** *Consider the following two literals:*

$$sys1,\ sys2$$

They can produce up to 4 possible configurations:

$$\begin{aligned} Configuration_1 &= sys_1 \wedge sys_2 \\ Configuration_2 &= sys1 \wedge \neg sys2 \\ Configuration_3 &= \neg sys1 \wedge sys2 \\ Configuration_4 &= \neg sys1 \wedge \neg sys2 \end{aligned}$$

So, for instance, if $Sys1 = (5 < x)$ *and* $Sys2 = (x < y)$*, then:*

$$\begin{aligned} Configuration_1 &= (5 < x) \wedge (x < y) = (5 < x) \wedge (x < y) \\ Configuration_2 &= (5 < x) \wedge \neg(x < y) = (5 < x) \wedge (x \geqslant y) \\ Configuration_3 &= \neg(5 < x) \wedge (x < y) = (5 \geqslant x) \wedge (x < y) \\ Configuration_4 &= \neg(5 < x) \wedge \neg(x < y) = (5 \geqslant x) \wedge (x \geqslant y) \end{aligned}$$

Internally, each literal (like sys_1 in Example 3.2) can represent either (1) a Boolean variable itself or, more interestingly (2) a numeric predicate. In our requirements, for the second case, we will consider numeric predicates of first order theories whose $\forall^*\exists^*$ fragment is decidable (for example, Presburger Arithmetic [Presburger, 1929]), but not undecidable theories like, for instance, Gödel arithmetic.

A tempting idea is to Booleanize a specification in which numeric predicates of one theory (say, theory of integers) are mixed with numeric predicates of another theory (say, theory of rationals). However, our current Booleanization does not allow these combinations, but only Booleanizations within predicates of the same type. In Section 4.1 the idea of mixing theories in the Booleanization is discussed as future work.

What is, indeed, allowed (and is explored in Subsection 4.2) is to *clusterize* the predicates that have relation between them, and compute their Booleanizations separatedly. This allows to separatedly Booleanize predicates of different theories in the same specification.

Potentials and antipotentials. Short names: pt_i and nt_i

We introduce now two concepts to capture the power that the system variable-owner has, after the environment moves: *potentials* and *antipotentials*.

Informally, a potential is a formula that states that a given configuration $conf_i(\bar{x},\bar{y})$ (which depends on the variables $\bar{x}$ belonging to the environment and the variables $\bar{y}$ belonging to the system) can be realized by some move of the system. In other words, after the environment moves $\bar{x}$, whether the system can play $\bar{y}$ and make $conf_i(\bar{x},\bar{y})$ hold. Formally, it is described in Definition 3.3 below.

***Definition* 3.3.** *Given $conf_i$, a potential $pt_i(conf_i)$ is a realizable configuration for the system. Formally, $pt_i(conf_i)$ is the following formula:*

$$\exists\bar{y} :: conf_i(\bar{x},\bar{y})$$

Note that the potential $\exists\bar{y} :: conf_i(\bar{x},\bar{y})$ has free variables $\bar{x}$ of the environment, which corresponds to those plays/moves of the environment after which the system can react by choosing to make $conf_i$ hold.

Therefore, we have to bear in mind that in order to decide whether a formula is realizable or not, its environment's variables have to be fixed. Thus, a potential is, in fact, an interpretation of a configuration which is done once the environment variables have been chosen. Once the variables are chosen, it is evaluated whether the combination of literals is realizable or not. So the real form of a potential (before fixing the $\bar{x}$) stands like this:

$$unbounded\ x(\exists\bar{y} :: configuration_i)$$

We can see that, if x is not fixed, then we will obtain a theory-equivalent formula without the $\bar{y}$ variables. In other words, $pt_i(conf_i)$ captures those x for which $\exists y\ :: conf_i$ hold. We can check it in Example 3.3 below.

Example **3.3.** *Let us take again the second configuration in Example 3.2: $sys1 \wedge \neg sys2$, which is $(5 < x) \wedge \neg(x < y)$. The resulting potential is:*

$$(\exists\bar{y} :: (5 < x) \wedge \neg(x < y))$$

where $\bar{y} = \{y\}$

In Presburger arithmetic this formula is equivalent to:

$$(5 < x)$$

This quantifier elimination has been performed using Cooper's algorithm, where the $\mathcal{A}$ set has been equal to $\{5\}$ and the $\mathcal{B}$ set has been equal to $\varnothing$.

The result are the same for $T_{\mathbb{R}}$, $T_{\mathbb{C}}$, but not for $T_{\mathbb{N}}$.

Similarly, antipotentials are the opposite concept of a potential: that is, an antipotential is a formula that states that a configuration $conf_i(\overline{x},\overline{y})$ cannot be realized by a move of the system (see Definition 3.4 below).

***Definition* 3.4.** *Given $conf_i$, an antipotential $nt_i(conf_i)$ is an unrealizable configuration for the system:*

$$\forall \overline{y} :: (\neg conf_i(\overline{x},\overline{y}))$$

which is equivalent to:

$$\neg \exists \overline{y} :: (conf_i(\overline{x},\overline{y}))$$

Note that, again, the $\overline{x}$ variables are free in $\forall \overline{y} :: (\neg conf_i(\overline{x},\overline{y}))$.

Therefore, if we apply the potential formula (or the antipotential -see Definition 3.4- formula that we will introduce now) formula to a configuration without having the valuations of the environment yet done, then the configuration is not intrinsically one or the other: it fully depends on the environment.

Potentials capture the power of the system player to react, while antipotentials state that the system is powerless.

Choosing some potentials can, later on, make sense or not for each player in a game, but all of them have to be considered in order to make the requirements equivalent to the original in terms of the power of each player. The antipotentials are not interesting for the system player (i.e. they will not appear in the Booleanized requirements), since they offer it somehow no *reaction* power.

Let us notice that for a given move of the environment (i.e. a choice of values of $\overline{x}$), some potentials may be realized, while others cannot, which is formally exemplified in Example 3.4 below.

Example **3.4** (Valid formula)**.** *Let $\overline{v}$ be a valuation of $\overline{x}$ and $conf_1(\overline{x},\overline{y})$ and $conf_2(\overline{x},\overline{y})$ be two configurations. The case described above (i.e some potentals being realized, while others not) occurs when the next formula is valid:*

$$(\exists \overline{y} \,.\, conf_1(\overline{x},\overline{y}) \wedge \forall \overline{y} \,.\, \neg conf_2(\overline{x},\overline{y}))[\overline{x} \leftarrow \overline{v}]$$

Note that the environment chooses variables first, and the the system *reacts* to those valuations with valuations of the system variables; which already gives us some clues that these requirements model a turn-based game.

Reactions. Short name: $react_i$ or r_i

In order to capture precisely the power that the system has for a given move of the environment, we introduce now the notion of *reaction*.

A reaction is a choice of some configurations to be potentials and the rest to be anti-potentials: that is, combinations of the configurations being potentials or anti-potentials. Formally:

***Definition* 3.5.** *Let $C \subseteq \mathcal{C}$ be a subset of the configurations. A reaction for C is defined as follows:*

$$react_C = \bigwedge_{c \in C} (pt(c)) \wedge \bigwedge_{c' \in \mathcal{C} \setminus C} (nt(c'))$$

Or, equivalently:

$$react_C = \bigwedge_{c \in C} (\exists \overline{y} \,.\, conf_c(\overline{x},\overline{y})) \wedge \bigwedge_{c' \in \mathcal{C} \setminus C} (\forall \overline{y} \,.\, \neg conf_{c'}(\overline{x},\overline{y})))$$

Note that $react_C$ has $\overline{x}$ as free variables, whereas it binds $\overline{y}$ in all its conjuncts.

Example **3.5.** *Consider the literals sys_1 and sys_2 used in Example 3.2 and their configurations. The following is an example of a reaction for $\mathcal{C} = \{conf_1, conf_2, conf_3, conf_4\}$:*

$$react_i = (\exists \overline{y} :: conf_1) \wedge (\exists \overline{y} :: conf_2) \wedge (\exists \overline{y} :: conf_3) \wedge (\forall \overline{y} :: \neg conf_4)$$

Note that some potentials/antipotentials combinations (i.e. some reactions) can never happen in the original numeric game, as not even a collaboration between the environment

and the system can make formula true. For instance, the formula $\exists a \in \mathbb{E}.\ \exists b \in \mathbb{S} :: (a > b) \wedge (a < b)$ is not satisfiable even with the collaboration of both players: therefore, reactions that have as potentials the configurations that take both literals positive can never be realized.

Combined reactions and combined valid reactions. Short names: φ_{React} and φ_{VR}

First of all, we have to distinguish reactions from a subset of it: valid reactions. They are described in Definition 3.7, after Definition 3.6 below.

***Definition* 3.6.** *We use react for the set of reactions, that is:*

$$react = \{c \in \mathcal{C} | react_C\}$$

***Definition* 3.7.** *A reaction r is called valid whenever there is a move of the environment for which r captures precisely the power of the system. Formally, a reaction is valid whenever* $\exists \overline{x}.\ r(\overline{x})$ *is valid. The set of valid reactions VR is defined as:*

$$VR = \{r \in React | \exists \overline{x}.\ r(\overline{x}) \text{ is valid}\}$$

We can now define the combined reaction (see Definition 3.8) and then define the combined valid reaction (see Definition 3.9) whose quantity is less or equal.

***Definition* 3.8.** *The* combined reaction *is the formula that states that all actions performed by the environment can be followed (i.e. responded) by some reaction of the system:*

$$\varphi_{React} = \forall \overline{x}. \bigvee_{r \in React} r$$

By inspection, we can see that φ_{React} is a valid formula.

***Definition* 3.9.** *Similarly to Definition 3.8 above, we define the* combined valid reactions *or just* valid reactions *(VR) as the formula that restricts the reactions that the system can follow to those that are valid:*

$$\varphi_{VR} = \forall \overline{x}. \bigvee_{r \in VR} r$$

Later in Theorem 3.1, we show that φ_{VR} is also a valid formula.

Also, remember that for the validity queries, we can make use of the quantifier elimination (see Subsection 2.3) where all the variables are bounded, but also other methods, some of which are gathered in the SMT-LIB [1] project (see Figure 3.1 below).

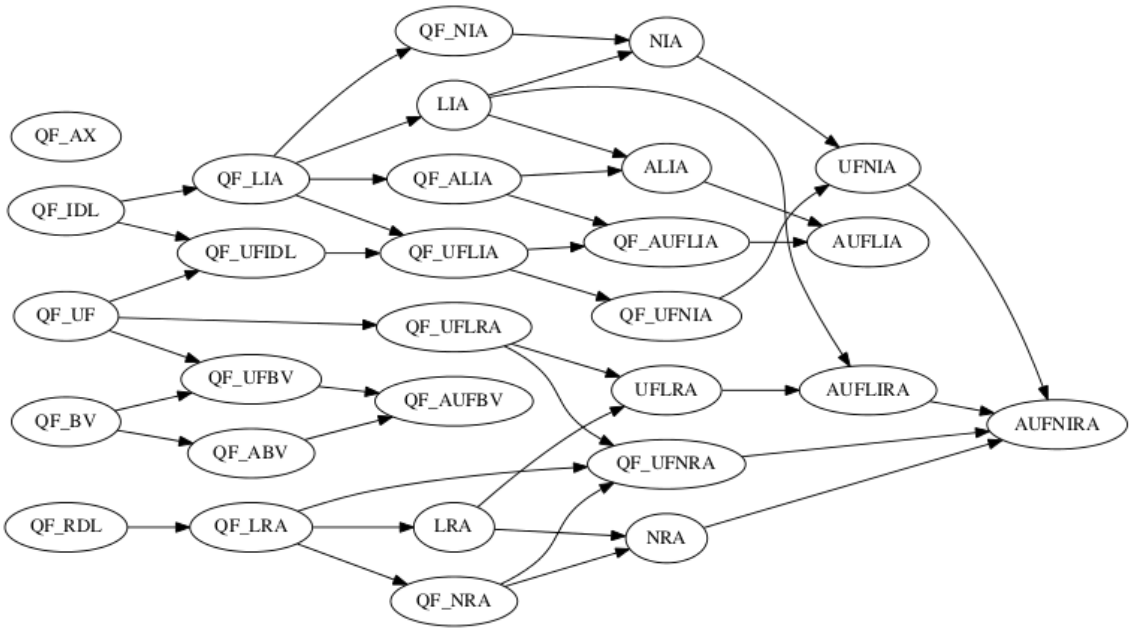

Figure 3.1: The diagram of the SMT Library, with some modules that solve quantified formulae.

Source: SMT Library.

The requirements Booleanization theorem

Given a set of formulae; for each of reaction, we can make the next validity query:

$$\exists \overline{x} :: react_i$$

where each conjunct in $react_i$, bounds all its $\overline{y}$ variables.

If the result is positive, then we include it in the set of reactions (i.e responses) that the system can perform. But we do not include the whole reaction, but only the potentials in the reaction, since it does not make any sense to the system to use the antipotentials (i.e. formulae that cannot realize, as stated in Definition 3.4).

Concretely, we include the potentials of the reaction to the set of configurations that belong to a new Boolean variable $\overline{e}$. This way, we associate to a fresh Boolean variable $\overline{e}_i$ in

[1] http://smtlib.cs.uiowa.edu/logics.shtml

the formula formed by the disjunction of the potentials in $react_i$, as follows:

$$\textit{if } (\exists \overline{x} :: react_i) \textit{ then, include the next requirement:}$$
$$fresh(\overline{e}) \implies pots(react_i)$$

where *pots* is a function that extracts the potentials of a reaction.

In other words, we obtain a conjuncted set of included requirements, that are pairs of the form $\bigvee_i (e_i, pots(r_i))$, where e_i is a decision of the environment that implies the, $pots(r_i)$ which are the potentials of a valid reaction for e_i. For further information, see Subsubsection 3.1.2.

We can depict this idea as an algorithm too (see Algorithm 1 below).

Algorithm 1: Brute-force Booleanization algorithm

$\varphi^{booleanized} \leftarrow \varphi^{Num}[l_i \leftarrow s_i]$;
(i.e. replacing theory-predicates with Boolean variables)
$\mathcal{C} \leftarrow$ obtain_the_configutations_from_the_literals;
$(\mathcal{P}, \mathcal{A}) \leftarrow$ From $\mathcal{C}$, compute the sets of potentials and antipotentials ;
$\mathcal{R} \leftarrow$ From $(\mathcal{P}, \mathcal{A})$, compute the reactions ;
Valid $\leftarrow \{\}$;
for *react in* $\mathcal{R}$ **do**
  if $\exists x :: (react_i)$ *is valid* **then**
    Valid $\leftarrow$ Valid $\cup$ $(e_i, pots(react_i))$;
  end
end
$\varphi^{extra} \leftarrow \bigvee_{(e,\rho)\in Valid} e \implies \bigwedge_{p\in\rho} p$;
$\varphi^{\mathbb{B}} \leftarrow \varphi^{booleanized} \wedge \varphi^{extra}$;
return $\varphi^{\mathbb{B}}$

*The algorithm is explained in Subsection 3.1.2.

This way, we have captured that for all plays of the environment, the system can always react, since it can react using at least one of the valid reactions. This is precisely given by φ_{VR}.

As said above, in Theorem 3.1, we show that φ_{VR} is a valid formula; and its consequences are presented in Claim 3.1 below.

Claim **3.1.** *For every formulae writen on a theory or fragment for which we can decide*

validity with one alternation of quantifiers (indeed, any $\exists^*\forall^*$ formulae), there is a correct Booleanization and it is computable by the Algorithm 1, where a correct Booleanization means that it preserves the exact power of each variable-owner in the original requirements.

The Claim 3.1 (which is proved in Subsection 3.2.1) above implies that there is a correct Booleanization, at least for the most typical arithmetical theories:

- For $\mathcal{T}_{\mathbb{Z}}$: That is, for Linear Integer Arithmetic, whose fast version we introduce in Subsection 4.1.
- For $\mathcal{T}_{\mathbb{Q}}$: That is, for Linear Rational Arithmetic, whose fast version we introduce in Subsection 4.1.
- For $\mathcal{T}_{\mathbb{R}}$: That is, for Nonlinear Real Arithmetic, whose fast version can be implemented with some version of the Cyndrical Algebraic Decomposition or the Virtual Substitution algorithms.

These last two are really relevant for real use industrial requirements modelling.

The corollary of Claim 3.1 is that games can be also Booleanizable, as stated in Claim 3.2.

Claim **3.2.** *Any game that is specified using a Booleanizable theory or fragment can have its equivalent game specified using only Boolean variables.*

Therefore, any game modeled using $\mathcal{T}_{\mathbb{Z}}$, $\mathcal{T}_{\mathbb{Q}}$ or $\mathcal{T}_{\mathbb{R}}$ can have its equivalent Booleanized game.

Note that Claim 3.2 needs more specific proofs that will be presented in Section 3.2.

Limits of Booleanization

Our Booleanization algorithm is restricted to literals that do not compare different instants of time. In other words, with our current LTL logic (see Subsection 2.1.3), nondeterminism sources like having temporal operators inside predicates are not allowed: for instance, $(a < \mathcal{Y}(b))$ is not a well-formed LTL formula, since $\mathcal{Y}(b) \in \mathbb{B}$ and $a \in Num$.

Another tempting idea is to incorporate a function $\mathcal{Q}$, that takes the value of a variable in a past timestep. This could allow to use predicates like $(x = \mathcal{Q}^2(x))$, which would mean *current timestep's x has the same value as two timesteps ago*. This idea is not covered by the current Booleanization, and it is explored in Subsection 4.1.

Therefore, the Booleanization technique is limited by the language and not by the algorithm itself: that is, the Booleanization technique's limits are exactly the limits of realizability checking: that is, the limits that the modal logic it is using has. In our case, we are limited by the expressivity of LTL.

3.1.2. Brute-force Booleanization Algorithm

Let us now see in more detail how the technique for Booleanization (see Algorithm 1) works. Its implementation in Ocaml is out of the scope of this thesis.

The formal description of the algorithm

The algorithms assumes the literals are given: that is, it has been called by some procedure that has isolated literals from the set of requirements. For instance, if $\varphi = (a < b) \wedge \mathcal{Y}(a \geqslant 1)$ is a specification, then, the algorithm receives the literals $(a < b)$ and $(a \geqslant 1)$.

The algorithm consists of six steps:

1. Convert theory-predicates to Boolean: each literal is associated to a fresh Boolean variable that belongs to the system and replaced in all requirements. For instance, $(a < b)$ is converted to $sys1$, no matter whether a and b belong to one player or the other.

2. Compute all the configurations of the literals, i.e. it computes all the combinations of the literals.

3. For all configurations, compute its permutations over the satisfiability of the system: that is, compute its potentials and antipotentials.

4. Using the the potentials and antipotentials, compute their combinations: that is, compute all the reactions.

5. If a reaction is satisfiable (i.e. there is a concrete valuation of the environment), then the potentials of the reaction are considered a *possible reaction* of the system to that valuation.

6. The Booleanized requirements are the original requirements with the literals replaced (step 1) plus the set of all the valuations with their possible reactions (i.e. the extra requirement -see Definition 3.10).

The key of this process is 5. step, so we have to focus on its interpretation (see Subsubsection 3.1.2). But before that, let us see an execution of Algorithm 1 for a simple case in Example 3.6 below.

Example **3.6.** *This example is interesting because it captures the power of each player.*

Let $\varphi = \Box(\varphi_{Req_0} \wedge \varphi_{Req_1})$*, where:*

$$Req_0 \equiv (x > 1000 \implies y \leqslant x) \wedge (x \leqslant 1000 \implies y > x)$$
$$Req_1 \equiv (y > x)$$

We will first (step 1) Booleanize φ*. We can realize that* $(x > 1000) \leftrightarrow (x \leqslant 1000)$ *and* $(y \leqslant x) \leftrightarrow (y > x)$*: that is, there are two contrary literals. Thus, there are only two literals:*

$$s_0 \equiv (x > 1000) \leftrightarrow (\neg s_0 \equiv (x \leqslant 1000))$$
$$s_1 \equiv (y \leqslant x) \leftrightarrow (\neg s_1 \equiv (y > x))$$

Then (step 2), we will obtain all the configurations:

$$conf_0 \equiv s_0 \wedge s_1$$
$$conf_1 \equiv s_0 \wedge \neg s_1$$
$$conf_2 \equiv \neg s_0 \wedge s_1$$
$$conf_3 \equiv \neg s_0 \wedge \neg s_1$$

For each configuration (step 3), the potential and antipotential have to be produced (or decided, in case they are a validity that yields true or false):

- *Configuration 0:*

$$\mathit{Pot} : \exists y.\ (x > 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x > 1000)$$
$$\mathit{Apot} : \neg\exists y.\ (x > 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x \leqslant 1000)$$

- *Configuration 1:*

$$\mathit{Pot} : \exists y.\ (x > 1000) \wedge (y > x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x > 1000)$$
$$\mathit{Apot} : \neg\exists y.\ (x > 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x \leqslant 1000)$$

- *Configuration 2:*

$$\mathit{Pot} : \exists y.\ (x \leqslant 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x \leqslant 1000)$$
$$\mathit{Apot} : \neg\exists y.\ (x > 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x > 1000)$$

- *Configuration 3:*

$$\mathit{Pot} : \exists y.\ (x \leqslant 1000) \wedge (y > x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x \leqslant 1000)$$
$$\mathit{Apot} : \neg\exists y.\ (x > 1000) \wedge (y \leqslant x) \textit{ which is } \mathcal{T}_{\mathbb{Z}} \textit{ equivalent to } (x > 1000)$$

To make calculations simple, we can see there are only two potentials and two antipotentials: $(x > 1000)$ *and* $(x \leqslant 1000)$*. Therefore, there can only be these combinations inside a reaction:*

$$\exists x.\ (x > 1000) \wedge (x > 1000) \textit{ which is True}$$
$$\exists x.\ (x > 1000) \wedge (x \leqslant 1000) \leftrightarrow (x \leqslant 1000) \wedge (x > 1000) \textit{ which is False}$$
$$\exists x.\ (x \leqslant 1000) \wedge (x \leqslant 1000) \textit{ which is True}$$

Now, for all the potential/antipotential permutations, we will compute the reactions (step 4) and see which of them are realizable. To do so, we will note down which are potentials with an upper $\mathcal{P}$ *and the antipotentials with an upper* $\mathcal{A}$*:*

- *Combination 0:*

$$\exists x.\ conf_0^{\mathcal{P}} \wedge conf_1^{\mathcal{P}} \wedge conf_2^{\mathcal{P}} \wedge conf_3^{\mathcal{P}} \leftrightarrow$$
$$\exists x.\ (x < 1000) \wedge (x < 1000) \wedge (x \leqslant 1000) \wedge (x \leqslant 1000)\ \textit{which is False}$$

- *Combination 1:*

$$\exists x.\ conf_0^{\mathcal{P}} \wedge conf_1^{\mathcal{P}} \wedge conf_2^{\mathcal{P}} \wedge conf_3^{\mathcal{A}} \leftrightarrow$$
$$\exists x.\ (x < 1000) \wedge (x < 1000) \wedge (x \leqslant 1000) \wedge (x \leqslant 1000)\ \textit{which is False}$$

- *Combination 2:*

$$We continue the process.$$

- *And we can see all the permutations are False, except for two: that is, the two in which all the four literals are the same. See them below:*

  - *Possible combination 1:*

$$\exists x.\ conf_0^{\mathcal{P}} \wedge conf_1^{\mathcal{P}} \wedge conf_2^{\mathcal{A}} \wedge conf_3^{\mathcal{A}} \leftrightarrow$$
$$\exists x.\ (x < 1000) \wedge (x < 1000) \wedge (x < 1000) \wedge (x < 1000)\ \textit{which is True}$$

  - *Possible combination 2:*

$$\exists x.\ conf_0^{\mathcal{A}} \wedge conf_1^{\mathcal{A}} \wedge conf_2^{\mathcal{P}} \wedge conf_3^{\mathcal{P}} \leftrightarrow$$
$$\exists x.\ (x \leqslant 1000) \wedge (x \leqslant 1000) \wedge (x \leqslant 1000) \wedge (x \leqslant 1000)\ \textit{which is True}$$

To finish, we will take the realizable reactions (step 5) and subtract their potentials as a possible reaction to a fresh environment variable:

$$\varphi_{extra} = fresh(e_0) \implies ((x < 1000) \wedge (x < 1000))\ \textit{which is } \mathcal{T}_{\mathbb{Z}}\ \textit{equivalent to } (x < 1000)$$
$$\wedge\ fresh(e_1) \implies ((x \leqslant 1000) \wedge (x \leqslant 1000))\ \textit{which is } \mathcal{T}_{\mathbb{Z}}\ \textit{equivalent to } (x \leqslant 1000)$$

So the final requirement (step 6) is the Booleanized predicates plus the extra requirement

of step 5:

$$\varphi^{\mathbb{B}} = \Box(req0_{Bool} \wedge req1_{Bool} \wedge \varphi_{extra}),$$

where $req0_{Bool} \equiv (s_0 \implies s_1) \wedge (\neg(s_0) \implies \neg(s_1))$ *and where* $req1_{Bool} \equiv \neg(s_1)$

Why the algorithm works: the extra requirement

With a single variable-owner, translating the numeric predicates to Boolean is an enough Booleanization, but when there are two variable owners it does not contribute anything if we want to preserve the original power of each of them. We need a stronger Booleanization that will capture those powers in a new requirement.

Thus, the reason why the Booleanization technique in Algorithm 1 is correct is that, contrary to the technique that will be shown in Subsubsection 3.1.3, we do not only Booleanizes the numeric predicates, but also preserves the original power of each variable-owner. This has been seen in Example 3.6.

These powers are modelled using the so-called extra requirements (see Definition 3.10 below), which is the union of all the *environment-valuation and its possible reaction* formulae computed in step 5.

***Definition* 3.10.** *The extra requirement* φ_{extra} *of the Booleanized specification* $\varphi^{\mathbb{B}}$ *of a numeric specification* φ^{Num} *is the formula that captures the original power relation between the variables of* φ^{Num}*. Formally:*

$$\varphi^{Num} \textit{ if and only if } \varphi^{\mathbb{B}}$$

where $(\varphi^{\mathbb{B}} = (\varphi_{booleanized} \wedge \varphi_{extra}))$

Computing a φ_{extra} is totally essential when there are two variable-owners in φ^{Num}.

Note that the usual shape of an extra requirement is as follows: a conjunction of environment decisions (i.e. $\overline{x}$'s valuations) each of them implying a disjunction of potentials of the reactions that were possible for the implicating decision. Formally, let $\mathcal{D}$ mean the set of decisions:

$$\bigwedge_{d \in D} [d \implies (\bigvee_{r \in VR} pot(r(d)))]$$

Or, in other words, by extension:

$$\begin{aligned}
&(dec = d_0 \implies (pot(r_0(d_0)) \vee pot(r_3(d_0)) \vee ...)) \wedge \\
&(dec = d_1 \implies (pot(r_2(d_1)) \vee ...)) \wedge \\
&(dec = d_2 \implies (...)) \wedge \\
&\quad ...
\end{aligned}$$

Note that, when it comes to the environment decisions, the Booleanization given by the extra requirements is not properly a Booleanization, but an enumeration on the decisions. So the *enum* type needs to be Booleanized. This is done in a very straightforward way, as in Example 3.7 below.

Example **3.7.** *Consider the following extra requirement:*

$$\begin{aligned}
&(dec = d_0 \implies (something) \wedge \\
&(dec = d_1 \implies (something) \wedge \\
&(dec = d_2 \implies (something)
\end{aligned}$$

Then, its bitwise Booleanization (see [Losada, 2020]) consists on treating the equivalence as a Boolean variable, and negating the rest of them. For instance, let $dec_0 \equiv dec = d_0$, $dec_1 \equiv dec = d_1$ *and* $dec_2 \equiv dec = d_2$, *then:*

$$\begin{aligned}
&(dec_0 \wedge \neg dec_1 \wedge \neg dec_2 \implies (something) \wedge \\
&(\neg dec_0 \wedge dec_1 \wedge \neg dec_2 \implies (something) \wedge \\
&(\neg dec_0 \wedge \neg dec_1 \wedge dec_2 \implies (something))
\end{aligned}$$

In summary: the extra requirement needs a bitwise Booleanization over the environment-decisions part so that the result is purely Boolean and does not contain enumerates.

Also, note that it can happen that some decisions share the same reactions (or, better said, the same potentials of the reactions). If this happens, then they can be collapsed in a single decision. Thus: if two (or more) environment decisions (i.e. valuations) share the same reactions, then they are modelling the same situations and, thus, they can be collapsed in a single one.

Last but not least, it is relevant to say that the complexity of the Algorithm 1 is exponential

in the number of literals, taking the quantifier-elimination executions as unitary. Concretely, it is $O(2^{2^l})$. This performance can be enhanced for real industrial requirements, at least for the cases identified in Subsection 4.1.

3.1.3. An incorrect method: using queries

We will now introduce a tempting incorrect method that does not produce correct Booleanization results, mainly because it does not produce the extra requirements as it is defined in Definition 3.10.

The query-based Booleanizaton

As in the bitwise Booleanization, the method we will explain now produces one fresh Boolean variable for each numeric predicate and that Booleanization defines the new requirements.

The way this is done is by (1) Booleanizing every numeric predicate, and then, (2) making a (set of) validity query(es) over each pair of fresh Boolean variable that substitutes the numeric predicate; to do so, both the variables of one element of the pair and the variables of the other are quantified. For each result of a query, we deduce different facts that are added as new requirements. The requirements that are added for each query can be seen in Algorithm 2.

As a corollary of the requirements added in Algorithm 2, one could say that if the first condition of it is fulfilled, we can make the incorrect Claim 3.3 below.

Claim **3.3.** *[**Incorrect** claim: realizability criteria for the query-based Booleanization method] Let $\mathcal{B}$ be the set of all the Booleanized predicates (i.e. literals).*

If for all pair of $\mathcal{B}$, an existential query over that variables that appear in the predicates results to be valid, then the following holds :

$$\textit{If} \bigwedge_{b_i,b_j \in \mathcal{B}} \exists \overline{v_{b_i}}, \overline{v_{b_j}} :: (b_i \wedge b_j) \textit{ then } (\textit{realizable}(\varphi^{Num}) \textit{ if and only if realizable}(\varphi^{\mathbb{B}}))$$

To see that, in some cases, Algorithm 2 above Booleanizes formulae correctly (and, indeed, the incorrect Claim 3.3 is fulfilled), we can perform an execution of it in Example 3.8 below.

Algorithm 2: The query-based Booleanization method as an algorithm

```
Booleanize_the_predicates;
(P ← obtain_the_predicate_pairs;
extra_reqs ← {};
for (pr1, pr2) in P do
    v_pr1 ← obtain_variables_from(pr1);
    v_pr2 ← obtain_variables_from(pr2);
    if ∃v_pr1, v_pr2 :: (pr1 ∧ pr2) is valid then
        extra_reqs ← extra_reqs ∪ ¬(pr1 ∧ pr2) ;
    end
    else
        if ∀v_pr1, v_pr2 :: (pr1 ∧ ¬pr2) ∨ (¬pr1 ∧ pr2) is valid then
            extra_reqs ← extra_reqs ∪ ((pr1 ⟹ pr2) ∨ (pr2 ⟹ pr1)) ;
        end
    end
end
```

Example **3.8.** *[A successful use case for the query-based Booleanization method]*

Let $\{a,b\}$ *be the variables of the system, and consider no environment variable. The numeric specification* $\varphi^{Num} = \square((a < b) \wedge (a \geqslant b) \wedge (a > b + 100)$ *is unrealizable.*

Also, let $q_1 \in \mathbb{B}$ *replace* $(a < b)$, $q_2 \in \mathbb{B}$ *replace* $(a \geqslant b)$ *and* $q_3 \in \mathbb{B}$ *replace* $(a > b + 100)$, *the three of them belonging to the system. Then:*

- $(q_1 \wedge q_2)$ *is invalid (i.e.* $\exists a,b.\ (a < b) \wedge (a \geqslant b)$*), therefore, we add:* $\neg(q_1 \wedge q_2)$
- $(q_1 \wedge q_3)$ *is invalid, therefore, we add:* $\neg(q_1 \wedge q_3)$
- $(q_2 \wedge q_3)$ *is valid, therefore, we make the next query:*
  - $(q_2 \wedge \neg q_3)$ *is valid, therefore, we add nothing.*
  - $(\neg q_2 \wedge q_3)$ *is invalid, therefore, we add:* $(q_3 \implies q_2)$

So the new requirements (i.e. the extra requirement) generated according to Algorithm 2 are:

1. $\neg(q_1 \wedge q_2)$
2. $\neg(q_1 \wedge q_3)$

3. $(q_3 \implies q_2)$

The realizability result of $\varphi^{\mathbb{B}} = \square(q_1 \wedge q_2 \wedge q_3)$ *in conjunction with the extra requirement is: unrealizable, since we cannot make* q_1 *and* q_2 *at the same time. Therefore, it has correctly Booleanized* φ^{Num}.

However, even if Example 3.8 above makes us think this method could replace Algorithm 1 presented in the previous Subsection 3.1.2, we will see it is not correct. The query-based Booleanization we are presenting is not recommendable, at least for two reasons that have been identified:

- It does not consider the different roles that the environment and the system play. This makes the method an *incorrect* Booleanization method. We will offer a counterexample in Subsubsection 3.1.3 below.
- It compares literals only in pairs. This makes the method inefficient. We will see an example in Subsubsection 3.1.3.

Each player's roles

As explained in Subsection 1.2, it is needed not only to Booleanize the numerical predicates, but also to compute a new requirement that will rightly model the relation between the fresh Boolean variables: that is, to correctly represent the power of each player via the variables he owns.

Without that relation, the original relation between the numerical variables is lost and, therefore, the Booleanized requirements are not equivalent to the original numeric ones.

This relation is provided by the definitive methods that can be seen in Subsection 3.1.1, but not with the query-based method in Algorithm 2. For instance, let us consider Example 3.9 below.

Example **3.9.** *[An example for the query-based Booleanization method]*

Let $\{x\}$ *be the variables of the environment,* $\{y\}$ *the variables of the system,* $b_1 \in \mathbb{B}$ *replace* $(y<x)$ *and* $b_2 \in \mathbb{B}$ *replace* $(y > 0)$*; where both* q_1 *and* q_2 *belong to the system.*

Also, let the numerical formula be $\varphi^{Num} = \square((y < x) \vee (y > 0))$*, which is realizable.*

Then:

1. *Now we make validity queries. We can verify* $\exists x,y.\ (y < x \wedge y > 0)$ *is valid and* $\exists x,y.\ (y < x \wedge \neg y > 0) \vee (y \geqslant x \wedge \neg y > 0)$ *is valid. As they are valid, then we keep this Booleanization of the requirements.*

2. *So the new formula stands like:*

$$\varphi^{\mathbb{B}} = \square(b_1 \wedge b_2)$$

Which is realizable.

But what if the original specification was:

$$\varphi^{Num} = \square((y < x) \wedge (y > 0))$$

φ^{Num} *is unrealizable, while its Booleanization* $\varphi^{\mathbb{B}}$ *created is realizable. Therefore, in this this example:*

$\bigwedge_{b_i,b_j \in \mathcal{B}} (b_i \wedge b_j)$ *holds, but* $(realizable(\varphi^{Num})$ *holds if and only if* $realizable(\varphi^{\mathbb{B}}))$ *does not hold*

Instead, the correct methods (see Subsection 3.1.1) would create a correct extra-requirement (see Definition 3.10) that models the power of the environment: $\varphi^{\mathbb{B}} = \square((b_1 \wedge b_2) \wedge (c \implies \neg(b_1 \wedge b_2))$*, where* $c \in \mathbb{B}$ *belongs to the environment.*

Note that in Example 3.9 above, since the system controls both Boolean variables, then there is no additional requirement as b_1 and b_2 can be freely chosen to be *true* of *false*, since all combinations are valid.

Thus, Example 3.9 above is a counter-example of the soundness of the query-based Booleanization method. This counter-example proves that Claim 3.3 is indeed incorrect.

Only considering pairs

When comparing pairs of predicates (performing existential validity queries), we are correctly evaluating the relationship between two predicates, but ignoring this relationship against the rest of the predicates. Example 3.10 below shows a simple case of this.

Example **3.10** (Comparing all the pairs separatedly is not correct)**.** *Let* $\varphi = (y < 3) \wedge (y > 1) \wedge (y \neq 2)$*, and let them be Booleanly represented* b_1*,* b_2 *and* b_3 *respectively, the three of them belonging to the system.*

We can see that every pair of literals is valid, but not all of them the same time. The query based method (see Algorithm 2) returns no extra requirement, while the correct one (see Algorithm 1) does generate:

$$\neg(b_1 \wedge b_2 \wedge b_3)$$

Note that the correct Booleanization methods generate the additional requirement without the need of any environment variables: that is, only the restriction $\neg(b_1 \wedge b_2 \wedge b_3)$ is generated.

To correct this problem, validity queries have to be made incrementally adding new predicates.

3.2. Correctness Proof of the Correct Algorithm

In this section we will provide the proof of correctness of the definitive solution for the Booleanization problem (see Subsection 3.1.1) and the algorithm (see Subsection 3.1.2) that solves it; both explained in the previous Section, 3.1.

To do so, two proofs are presented:

1. *The Local Booleanization Theorem* ensures that the Brute-Force Booleanization Algorithm (see Subsection 3.1.2) over numeric requirements Booleanizes them correctly: that is, Algorithm 1 outputs Boolean requirements that correctly model the original numeric power of each variable-owner (i.e. player if we talk about games). This theorem is presented in the immediately next Subsection 3.2.1.
2. *The Temporal Booleanization Theorem* ensures that, if the correctly Booleanized requirements model a temporal game, then the Booleanized game correctly models the power of each player in the original numeric game; which is the same as saying that the original numeric specification and the Boolean specification are equi-realizable. This theorem is presented in Subsection 3.2.2.

The second theorem leans on the first one.

3.2.1. The Local Booleanization Theorem

The theorem states the equivalence relationship between Boolean and numeric requirements (ie. formulae).

For the moment, no temporality or game concepts are introduced in this subsection; which will be used in Subsection 3.2.2.

The Reaction Existence Lemma

This lemma is essential for the main statement of the Local Booleanization Theorem that will be explained immediately next in Subsubsection 3.2.1 leans on it.

It states that every move of the environment, can be followed by a move of the system. This is precisely stated in Lemma 3.1.

Lemma **3.1** (Reaction existence lemma)**.** *For every valuation of $\bar{x}$, $\bar{v}$, there is at least one reaction r such that $r[\bar{x} \leftarrow \bar{v}]$ is valid. Formally, $\varphi_{React} = \forall \bar{x}. \bigvee_{r \in React} r$ is valid.*

Proof. We need to show that $\varphi_{React} = \forall \bar{x}. \bigvee_{r \in React} r$ is valid.

Let $\bar{v}$ be an arbitrary valuation of the variables $\bar{x}$ (i.e. a choosing-move of the environment), and let $C = \{c \in conf \mid I[\bar{x} \leftarrow \bar{v}] \models \exists \bar{y}.\ conf(\bar{y})\}$.

It follows that $I[\bar{x} \leftarrow \bar{v}] \models react_C$, since for every $c \in C$ then $I[\bar{x} \leftarrow \bar{v}] \models c$ and for any $c \notin C$ then $I[\bar{x} \leftarrow \bar{v}] \not\models c$.

□

The Valid Reaction Existence Theorem

We will explain the main local theorem: that is, that for every movements of the environment, there is a valid reaction of the system. This is the same idea as the Claim 3.1 that has been stated in Subsubsection 3.1.1, and is properly stated in Theorem 3.1.

Theorem **3.1.** *For every movement of the environment, the system can move at least with one of the stored (i.e. valid) reactions. Formally, $\varphi_{VR} = \forall \bar{x}. \bigvee_{r \in VR} r$ is a valid formula.*

Proof. By contradiction, assume $\varphi_{VR} = \forall \bar{x}. \bigvee_{r \in VR} r$ is not valid.

Then, there is an interpretation I, that does not realize φ_{VR}. Formally: $I \not\models \varphi_{VR}$; or, equivalently: $I[\bar{x} \leftarrow \bar{v}] \not\models \bigwedge_{r \in VR} r$, for some $\bar{v}$.

Now, by Lemma 3.1, we know φ_{React} is valid: so, $I[\bar{x} \leftarrow \bar{v}] \models \bigwedge_{r \in React} r$, for some $\bar{v}$: that is, there is a reaction r such that $r[\bar{x} \leftarrow v]$ is valid. Therefore, $\exists \bar{x}.r$ is valid, so $r \in VR$. This means: $I[\bar{x} \leftarrow v] \models r$.

It follows that there is a $r \in React \setminus VR$ such that $I[\overline{v} \leftarrow \overline{x}] \models r$, which implies that $I \models \exists \overline{x}.\ r$.

Since $I \models \exists \overline{x}.\ r$ is closed, $I \models \exists \overline{x}.\ r$ is valid.

This is a contradiction.

□

Interpretation of the Valid Reaction Existence Theorem

As an observation, in the extra requirement, the set of potentials in valid reactions cannot be empty. In other words, for every move of the environment the system can always move with a valid reaction, which will result in the always-existence of some outcome. This is stated in Lemma 3.2.

Lemma **3.2.** *Let $C \in \mathcal{C}$ be such that $react_C \in VR$. Then potentials $C \neq \varnothing$.*

Proof. Bear in mind $react_C \in VR$ is valid.

Let $\overline{v}$ be such that $react_C[\overline{x} \leftarrow \overline{v}]$ is valid. Let $\overline{w}$ be an arbitrary valuation of $\overline{y}$. Let $conf$ be a configuration and let l be a literal.

Therefore:

$$\bigwedge_{l[\overline{x} \leftarrow \overline{v}, \overline{y} \leftarrow \overline{w}] \text{ is true}} l \wedge \bigwedge_{l[\overline{x} \leftarrow \overline{v}, \overline{y} \leftarrow \overline{w}] \text{ is false}} \neg l$$

It follows that $I[\overline{x} \leftarrow \overline{v}] \exists \overline{y}.\ c$, so $c \in C$.

□

This Lemma 3.2 is crucial, because it ensures that once the Brute-Force Booleanization algorithm (see Subsection 3.1.2) is executed, for each fresh $\overline{e}$ variable in the extra requirement, at least one reaction with one or more potentials on it can be responded by the system.

The interpretation for this is that, if we know that for each $\overline{x}$ there is a reaction (i.e. a discretization or enumeration of $\overline{y}$), then we can group (i.e. discretize or enumerate) all potentials of the set of possible $\overline{x}$ in a disjunction.

3.2.2. Temporal Booleanization Theorem

This theorem (which will be stated later properly in Theorem 3.1) is the main contribution of the thesis.

The theorem states the realizability equivalence between a game given by some specifications $\varphi^{\forall^*\exists^*}$ of a decidable $\exists^*\forall^*$ fragment of a logic, and the game given by some Boolean $\varphi^{\mathbb{B}}$ specifications that have resulted from the Booleanization algorithm described in Subsection 3.1.2.

Note that, as a witness of the fragment, we will be talking about an arbitrary *Num* numeric theory that has a a decidable $\exists^*\forall^*$ fragment (for instance, the theory of integers or the theory of rationals). That is, if we re-state the previous paragraph: The theorem states the realizability equivalence between a game given by some numerical φ^{Num} specifications, and the game given by some Boolean $\varphi^{\mathbb{B}}$ specifications that have resulted from the Booleanization algorithm described in Subsection 3.1.2.

Time and games

In both the Boolean and the numeric game, we will be distinguishing the *timesteps* (see Definition 3.11), *trace length* (see Definition 3.12) and *temporal depth* (see Definition 3.12) concepts.

***Definition* 3.11.** *We define a timestep as the temporal points that a computation of a temporal-specification-given formula can traverse, from* 0 *to potentially* ω.

Note that in each timestep, each variable-owner chooses his variables. This is the same as saying that, if we had a temporal formula φ and a timestep k, then the choice of variable-owners in k yields a subformula φ_k constituent of φ.

Also, note that a timestep is incremented once both variable-owners have chosen their variables. This yields an unrolling constituent formulae of the original formula with the form: $\varphi = \varphi_0 \wedge \varphi_1 \wedge \ldots$

In Figure 3.2 below, we can see how all the ψ subformulae are timesteps previous to the φ.

We can easily see the difference with the trace length in Definition 3.12 below.

***Definition* 3.12.** *The Trace length is the number of timesteps (or unrollings) that a temporal formula has evaluated since the timestep number* 0.

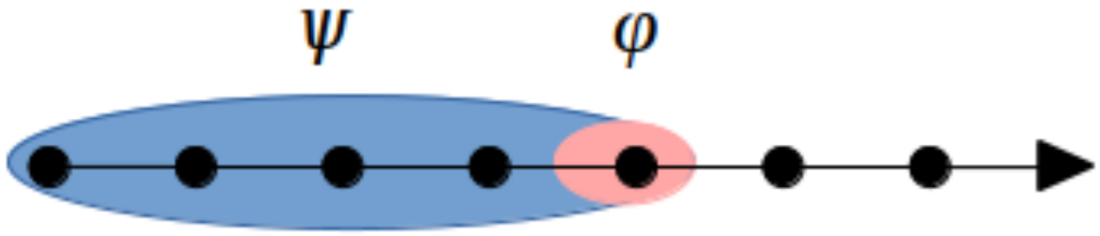

Figure 3.2: A diagram with timesteps, where we can see all the ψ subformulae (each point) are timesteps previous to the φ.

Source: Source: Pg 13 in [Losada, 2020]

Note that the index that represents the current point of the evaluation of a temporal formula (i.e. the timestep) denotes also its trace length. This is given by the computation of the game.

Example **3.11.** *Consider the following specification:*

$$\phi = \Box\varphi, \text{ where } \varphi = \mathcal{R}_1 \wedge \mathcal{R}_2 \wedge \mathcal{R}_3$$

Where:

$$\begin{aligned} \mathcal{R}_1 : i_1 &\implies \mathcal{Y}(v_1) \\ \mathcal{R}_2 : v_1 &\implies o_1 \\ \mathcal{R}_3 : \mathcal{Y}(i_2) &\implies o_2 \end{aligned}$$

Where $i1 \in \mathbb{E}$ *and* $i2 \in \mathbb{E}$*, and where* $v1 \in \mathbb{S}$ *and* $o1 \in \mathbb{S}$ *and* $o_2 \in \mathbb{S}$*.*

Then, if we are in timestep 1 *the trace length is* 2*, where each timestep on the trace is a concrete unrolling of* φ*:*

- *Unrolling* 1 *(timestep* 0*):*

$$\begin{aligned} \mathcal{R}_1 : i_{1_{t_0}} &\implies v_{1_{t_{0-1}}} \\ \mathcal{R}_2 : v_{1_{t_0}} &\implies o_{1_{t_0}} \\ \mathcal{R}_3 : i_{2_{t_{0-1}}} &\implies o_{2_{t_0}} \end{aligned}$$

Note that $0-1$ *is* -1*, which means that the variable's valuation falls off the trace (that is, that there is no timestep* t_{-1}*); thus, its value, by properties of Past LTL (see Subsubsection 2.1.3), is True.*

- *Unrolling* 2 *(timestep* 1*):*

$$\begin{aligned} \mathcal{R}_1 : i_{1_{t_1}} &\implies v_{1_{t_0}} \\ \mathcal{R}_2 : v_{1_{t_1}} &\implies o_{1_{t_1}} \\ \mathcal{R}_3 : i_{2_{t_0}} &\implies o_{2_{t_1}} \end{aligned}$$

Note that there is no more variable-valuation falling off the trace. Instead, the variables' valuations reference the previous timestep, which can be remembered.

The formula up to timestep 2 *is the conjunction of both unrollings.*

**A deep analysis of the unrolling concept has been performed in an alternative research line, based in [Arteche and van der Hallen, 2020].*

The remaining related essential concept is the *temporal depth* (see Definition 3.13 below), which has to be distinguished, mainly, from the concept of *timestep* (see Definition 3.11).

***Definition* 3.13.** *The temporal depth represents the number of previous steps that the evaluation must use: that is, the maximum order (i.e. the maximum quantity of nested* $\mathcal{Y}$*) of a* $\mathcal{Y}$ *operator.*

Note that the temporal depth is directly given by the specification. For instance, the formula $a = \mathcal{Y}^4 b \wedge \mathcal{Y} c$ has temporal depth 4 (its highest temporal order). Also, note that we can refer to the temporal depth of a single literal in the same way: the quantity of nested $\mathcal{Y}$. For instance, the temporal depth of $\mathcal{Y}^3(\phi)$ is 3.

Once again, note that trace length and the temporal length concepts are not the same: a trace length (equivalent to the current timestep number) is the number of timesteps since the computation began; while the temporal length of a formula φ denotes how many previous steps (before i) of the trace σ are needed to evaluate whether $(\sigma, i) \models \varphi$. Informally said, it denotes how much of the trace length can the evaluation remember (i.e. the maximum order of a $\mathcal{Y}$ operator).

To finish, when a literal makes reference to a timestep (see Definition 3.11) that has does not exists (say t_k, where $k < 0$) then the result is equivalent to $\mathcal{Y}(\bot) = \top$ (see Lemma 2.2).

In practice, this can happen when the temporal depth of a literal is greater than the number of timesteps (or the trace length). This trivial-truth property is described in Definition 3.14 below.

***Definition* 3.14.** *[Falling off the trace] Given a trace σ, when there is some literal which is operated with a nested quantity k of $\mathcal{Y}$ and k is strictly greater than the trace length $|\sigma|$, then that literal's verdict is $\top$ and we say it falls off the trace.*

Then, for instance, if $|\sigma| = 2$, then $\mathcal{Y}(\phi)$ has to be evaluated, while $\mathcal{Y}^4(\phi')$ is directly equal to $\top$.

In other words, if we see an LTL specification as a syntactic tree (see Example 3.12 below), all the literals whose quantity of nested $\mathcal{Y}$ (see Definition 3.13) is greater than the current trace length are evaluated to $\top$. Let us see

Example **3.12.** *[An LTL specification as a tree] Let us consider the following LTL specification:*

$$\psi = \phi_1 \wedge (\mathcal{Y}(\phi_2) \vee \mathcal{Y}^2(\phi_3))$$

If we depict is as a syntactic tree, we obtain Figure 3.3 below.

A tree-form LTL specification with a given trace, where the red points are subformulae and where the subformulae withing the green area denote those subformulae that fall off the trace (i.e. whose literals are all true).

We can see all the literals that belong to subformulae where its temporal depth is greater than $|\sigma|$ (i.e. they fall off the trace) are $\top$. That is:

- *Let ϕ_1 : $a \wedge \neg b$, then both a and b have to be evaluated.*
- *Let ϕ_2 : $a \wedge \neg b$, then both a and b have to be evaluated in the previous timestep.*
- *Let ϕ_3 : $a \wedge \neg b$, then both a and b fall off the trace, so their truth value is $\top$. Note that ϕ_3 does not have the $\top$ value (indeed, it is $\bot$), but its literals.*

Note that this same tree-construction is be used further when proving the Temporal Theorem in Lemma 3.5.

We will now introduce the essential components of our numeric and Boolean game in a top-bottom approach: (1) from games to arenas, (2) from arenas to positions (and their semantics), and (3) from positions to transitions.

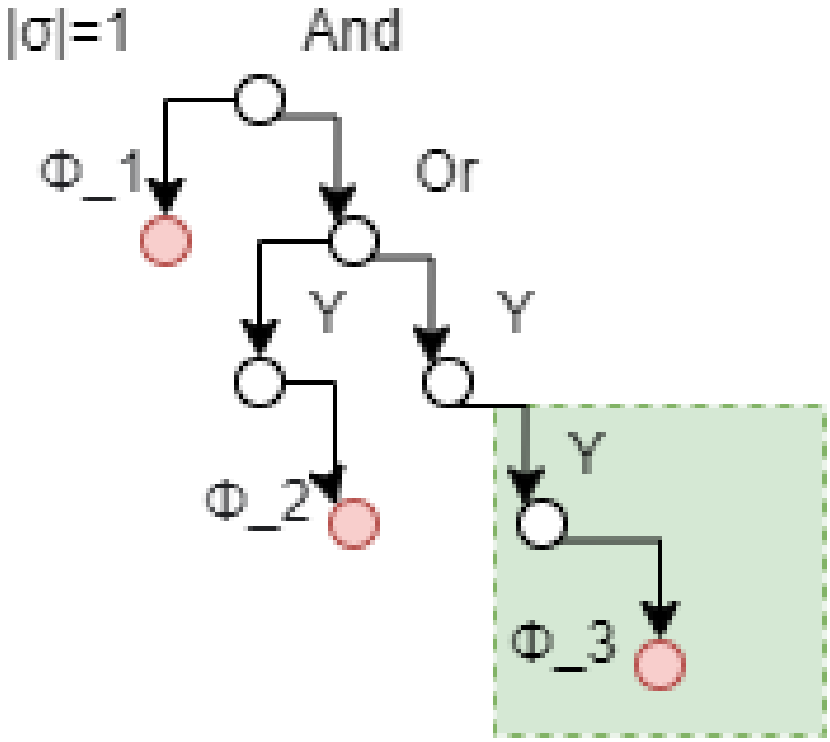

Figure 3.3: A tree-form LTL specification with a given trace, where the red points are subformulae and where the subformulae withing the green area denote those subformulae that fall off the trace (i.e. whose literals are all true).

Source: Own

Formal description of the numeric game

Numeric game overall. Short name: $\mathcal{G}^{Num}$

Given a numerical specification φ we construct a safety *numeric* game $\mathcal{G}_{\varphi}^{Num}$ such that there is a system that can model-check the specification φ whenever the system player can win the game.

The game is described as follows in Definition 3.15 below, which is nothing more than an instantiation of the definition of a safety game (see Definition 2.10) with the notation and clarifications that will be used during this subsection

***Definition* 3.15.** *The numeric game $\mathcal{G}^{Num}$ is composed of an arena $\mathcal{A}^{Num}$ and a set of winning positions Win^{Num}, where the arena contains a set of initial positions $\mathcal{I}^{Num}$ (we will feel free to call them also states as a synonym), a set of positions $\mathcal{V}^{Num}$ and a set of*

transitions $\mathcal{T}^{Num}$. Formally:

$$\mathcal{G}^{Num} = (\mathcal{A}^{Num}, Win^{Num})$$

where $\mathcal{A}^{Num} = \{\mathcal{I}^{Num}, \mathcal{V}^{Num}, \mathcal{T}^{Num}\}$ are described below.

During this subsubsection, 3.2.2, we drop the subindex referring to the specification φ whenever it is clear from the context. We now describe the arena, positions, moves and winning condition of the game.

Numeric arena. Short name: $\mathcal{A}^{Num}$

Independently of the kind of numeric theory we will use, the arena $\mathcal{A}^{Num}$ of the game $\mathcal{G}^{Num}$ can be depicted as a graph.

Because of the LTL (see Subsection 2.1.3) nature of the requirements we are dealing with, the arena is a static frame of the game whose structure in terms of states does not change: $\text{env}_{t_0} \rightarrow \text{sys}_{t_0} \rightarrow \text{env}_{t_1} \rightarrow \text{sys}_{t_1} \rightarrow ...$, which produce the arena $\mathcal{A}^{Num}$ in timestep 0, $\mathcal{A}^{Num}_{\varphi^{t_0}}$; the arena $\mathcal{A}^{Num}$ in timestep 1, $\mathcal{A}^{Num}_{\varphi^{t_1}}$; etc.

The arena $\mathcal{A}^{Num}$ contains a set $\mathcal{I}^{Num}$ of initial positions, which represent that we are entering (or *initializating*) $\mathcal{A}^{Num}$. Concretely, in terms of a temporal game, $\mathcal{I}^{Num}$ usually serves to connect the current k timestep's $\mathcal{A}^{Num}_{\varphi^k}$ with the previous k timestep's $\mathcal{A}^{Num}_{\varphi^{k-1}}$. Since this information is not relevant for this description, we will ignore the $\mathcal{I}^{Num}$.

Thus, $\mathcal{A}^{Num}$ has a concrete structure, that we can find in Definition 3.16.

***Definition* 3.16** (Numeric Arena)**.** *The arena $\mathcal{A}^{Num}$ is a graph $\langle \mathcal{S}, \mathcal{T} \rangle$, where:*

- *$\mathcal{S} = \mathcal{S}_e \cup \mathcal{S}_s \cup \mathcal{S}_{val}$ is the set of positions, which is composed of:*
  - *$\mathcal{S}_e$ is the set of positions that belong to the environment, that is, from which the valuations of the environment variables are made.*
  - *$\mathcal{S}_s$ is the set of positions that belong to the system: (1) they contain the valuations of the environment and (2) give valuations to the system variables.*
  - *$\mathcal{S}_{val}$ is the set of positions that are evaluable: they note down all the valuations made on the current timestep and use the ones also stored from the previous instants (up to the maximum time order) and evaluate the specification. After this state, a clock-tick is done: that is, we move to the next timestep.*

- $\mathcal{T}$ *is the set of transition between positions: that is, choosing variables in the numeric game* $\mathcal{G}^{Num}$.

Note that, as just said, the arena $\mathcal{A}^{Num}$ of Definition 3.16 gets rid of the $\mathcal{I}^{Num}$ used in Definition 3.15.

Also note that $\mathcal{A}^{Num}$ is inherently finite, since the temporality produces ω *copies* of it: the above mentioned $\mathcal{A}^{Num}_{\varphi^{t_0}}$, $\mathcal{A}^{Num}_{\varphi^{t_1}}$, ..., $\mathcal{A}^{Num}_{\varphi^{t\omega}}$

On the other side, each $\mathcal{A}^{Num}_{\varphi^k}$ can be finite or infinite, in the sense that the decisions of each player result either in a finite or an infinite number of transitions, as Example 3.13 below shows.

Example **3.13.** *Consider a literal* $l = (x > 10)$*, where* x *belongs to the environment:* $x \in \mathbb{E}$.

Assume now l *is the only literal of the game that contains environment variables on it. Thus, the environment can choose between infinite values of* x *that satisfy* l*, say:* $x = 11, x = 12, \ldots$ *This leads in an infinite number of transitions (and states) in* $\mathcal{A}^{Num}_{\varphi^k}$ *(i.e. we can depict it with a finite frame).*

If, on the contrary, there was another literal that constrains the value of x *to a finite number of positions, then the frame would be finite. This could happen, for instance, if we added a new literal* $l_2 = (x \leqslant 15)$*, or another more restrictive one* $l_3 = (x = 11)$

On the other hand, let us consider some positions as *Good* positions and some positions as *bad* positions, the latter being defined in Definition 3.17 below.

***Definition* 3.17.** *Bad positions* $\mathcal{B} \subseteq \mathcal{S}_{Eval}$ *are the evaluable positions in an arena* $\mathcal{A}^{Num}$*, where, given a set of current and previous (up to the temporal depth) valuations, their evaluation (or verdict) has been false or* $\perp$.

On the contrary, we know that : $Good = \mathcal{S} \backslash \mathcal{B}$

Numeric positions. Short name: $\mathcal{V}^{Num}$

We can now describe what a position is.

Positions not only carry the information about what the player can play, but also some history, so not only the player but also the history must be noted down. This is properly explained in Definition 3.18.

***Definition* 3.18.** *Let d be the temporal length of a play, and ρ a certain player (i.e. the environment or the system).*

A current position $\mathcal{V}_\rho^d$ in a numeric game $\mathcal{G}^{Num}$ is a tuple that contains a bit which controls if it is violating the specification, together with the sequence of valuations of the environment and the system up to a maximum temporal length $k \leqslant d$. Formally:

$$\mathbb{B} \times (\mathbb{N}^n \times \mathbb{N}^m)^{\leqslant k}$$

where $(\mathbb{N}^n \times \mathbb{N}^m)^{\leqslant k} = (\mathbb{N}^n \times \mathbb{N}^m)^0 \cup (\mathbb{N}^n \times \mathbb{N}^m)^1 \cup \ldots \cup (\mathbb{N}^n \times \mathbb{N}^m)^k$

We now also describe the set of positions, in Definition 3.19 below.

***Definition* 3.19.** *The set of positions is the union of the positions that belong to the environment, the positions that belong to the system and evaluable positions. Formally:*

$$\mathcal{V} \subseteq \mathcal{V}_{Env} \cup \mathcal{V}_{Sys} \cup \mathcal{V}_{Eval}$$

Now, depending on the kind of position, the position contains different levels of information: (1) the environment ones will only have the history, (2) the system ones will have the history plus the current decision of the environment in the current timestep, and (3) the evaluable positions contain the history plus both the decisions of the environment and the system in the current timestep (i.e the whole history up to the current temporal length). Formally:

- $\mathcal{V}_{Env}$: $\mathbb{B} \times (\mathbb{N}^n \times \mathbb{N}^m)^k$
- $\mathcal{V}_{Sys}$: $\mathbb{B} \times (\mathbb{N}^n \times \mathbb{N}^m)^k \times \mathbb{N}^n$
- $\mathcal{V}_{Eval}$: $\mathbb{B} \times (\mathbb{N}^n \times \mathbb{N}^m)^k \times (\mathbb{N}^n \times \mathbb{N}^m) = \mathbb{B} \times (\mathbb{N}^n \times \mathbb{N}^m)^{\leqslant k+1}$

The control bit $b \in \mathbb{B}$, will be used to record whether the specification has been violated (i.e. reached a bad state -see Definition 3.17-), but it can be modified to represent different violations, as the one we will see when defining transitions' properties in Lemma 3.3.

Semantics of the numeric game. Short name: Sem^{Num}

We now describe in Definition 3.20 below how to evaluate the realizability of an arbitrary position: that is, define the semantics of an arbitrary $(\mathbb{N}^n \times \mathbb{N}^m)^i \models \varphi$, taking into account φ is a Past LTL formula (see Subsubsection 2.1.3).

***Definition* 3.20** (Definition of semantics)**.** *Let p be 'an atom' and $\phi = \boxminus\varphi$, and $\sigma = ((\bar{x}_0,\bar{y}_0),(\bar{x}_1,\bar{y}_1),...,(\bar{x}_j,\bar{y}_j))$ be the trace given by the valuations of the environment and the system in each timestep. Also, let $\sigma^{-1} = ((\bar{x}_0,\bar{y}_0),(\bar{x}_1,\bar{y}_1),...,(\bar{x}_{j-1},\bar{y}_{j-1}))$ be the trace on the previous timestep of σ. We can define the finite semantics Sem^{Num} of $\mathcal{G}^{Num}$ as follows:*

$$\begin{array}{llll} \sigma & \models^{fin} & p & \text{iff} \quad p(\bar{x}_j,\bar{y}_j),\ \text{where } 0 \leqslant j \leqslant k \\ \sigma & \models^{fin} & \varphi_1 \wedge \varphi_2 & \text{iff} \quad \sigma \models^{fin} \varphi_1 \text{ and } \sigma \models^{fin} \varphi_2 \\ \sigma & \models^{fin} & \neg\varphi & \text{iff} \quad (\sigma \not\models^{fin} \varphi) \\ \sigma & \models^{fin} & \mathcal{Y}\varphi & \text{iff} \quad \text{either } |\sigma| = 1 \text{ or } \sigma^{-1} \models \varphi \end{array}$$

Note that the rest of the operators are derived from these (such as $\vee$), and $\mathcal{Y}$ is the classical *Yesterday* explained in Subsection 2.1.3.

We can see that an infinite trace $\sigma \models \boxminus\varphi$, where φ is Past LTL, if and only if for every k: $(\sigma_{k-d},...,\sigma_k) \models^{fin} \varphi$

Transitions in the numeric game. Short name: $\mathcal{T}^{Num}$

We can, finally, describe the set of transitions in Definition 3.21.

***Definition* 3.21** (Set of transitions)**.** *The set of transition consists on the union of the transitions of the environment, the transitions of the system and the so-called evaluable transitions (i.e. transition from evaluable positions to environment positions). Formally:*

$$\mathcal{T} \subseteq T_{env} \cup T_{sys} \cup T_{eval}$$

where $T_{env} \subseteq (\mathcal{V}_e \times \mathcal{V}_s)$, $T_{sys} \subseteq (\mathcal{V}_s \times \mathcal{V}_s)$ and $T_{eval} \subseteq (\mathcal{V}_s \times \mathcal{V}_e)$

We can now describe the properties of each element of this set in Definition 3.22 below.

***Definition* 3.22** (Definition of T_{env}, T_{sys} and T_{eval})**.** *Let the function removeLast be a fun-*

ction that removes the farthers temporality´s valuations. Now, for each transition type:

$$T_{env}(v,v') \textit{ holds when } v=(b,h) \textit{ and } v'=(b,h,\bar{x}) \textit{ for some } \bar{x}\in\mathbb{N}^n$$
$$T_{sys}(v,v') \textit{ holds when } v=(b,h,\bar{x}) \textit{ and } v'=(b,h::(\bar{x},\bar{y})) \textit{ for some } \bar{y}\in\mathbb{N}^m$$
$$T_{eval}(v,v') \textit{ holds when } v=(b,h) \textit{ and } v'=(b,removeLast(h)), \textit{ where } b'=true \textit{ iff } h\models\varphi$$

Note that we can also define T_{eval}, so that the b bit at *false* value does not mean *the spec is violated at this timestep*, anymore, but: *the spec has been violated at some point of the history*. This would be as it follows in Definition 3.3.

Lemma **3.3.** *Let removeLast mean the same as in Lemma 3.22. Now, the properties of each transition type would be:*

$$T_{eval}(v,v') \textit{ holds when } v=(b,h) \textit{ and } v'=(b,removeLast(h)),$$
$$\textit{where } b'=true \textit{ iff } b_{histo} \textit{ and } h\models\varphi$$

where b_{histo} has been initialized to true at some point and represents the verdict value of the game: $\top$ or $\bot$.

Lemma 3.3 above is essential, since this is the definition of T_{eval} that the games constructed by us will use (see Subsubsection 3.2.2).

Formal description of the Boolean game

This part will be the analogous to the immediately preceding Subsubsection 3.2.2 applied to the Boolean game.

Boolean game overall. Short name: $\mathcal{G}^{\mathbb{B}}$

The structure of the game $\mathcal{G}^{\mathbb{B}}$, contructed from $\varphi^{\mathbb{B}}$, is analogous to the $\mathcal{G}^{Num}$ Definition 3.15. We will state in formally in Definition 3.23.

***Definition* 3.23.**

$$\mathcal{G}^{\mathbb{B}}=(\mathcal{A}^{\mathbb{B}},Win^{\mathbb{B}})$$

where $\mathcal{A}^{\mathbb{B}}=\{\mathcal{I}^{\mathbb{B}},\mathcal{V}^{\mathbb{B}},\mathcal{T}^{\mathbb{B}}\}$ and $\text{Win}^{\mathbb{B}}=\mathcal{B}^{\mathbb{B}}$

Boolean arena. Short name: $\mathcal{A}^{\mathbb{B}}$

The arena (now finite frame) $\mathcal{A}^{\mathbb{B}}$ represents the requirements once they have been properly Booleanized.

It is structurally analogous to the arena $\mathcal{A}^{Num}$ seen before in Definition 3.16, except for two modifications in the states:

- There is a new type of position, called sink position $\mathcal{V}_{Sink}$, which is described in Definition 3.1.
- The number of position (produced by the transitions) is not infinite now: indeed, each environment-system position pair of the Boolean arena $\mathcal{A}^{\mathbb{B}}$ represents a concrete set of the environment-system pair of the numeric arena $\mathcal{A}^{Num}$.

Description **3.1.** *A sink position $\mathcal{V}_{Sink}$ captures valuations of the environment and system variables do not satisfy the extra requirement φ_{extra} given when the Booleanization of the original numeric requirements.*

Note that sink positions can only occur in Boolean games, since there is no φ_{extra} that can be violated in a numeric one.

Boolean positions. Short name: $\mathcal{V}^{\mathbb{B}}$

In $\mathcal{G}^{\mathbb{B}}$, the positions are described exactly in the same way as in $\mathcal{V}^{Num}$, seen in Definition 3.18, except that the definition of the *sink positions* $\mathcal{V}_{Sink}$ is added (a sink position $\mathcal{V}_{Sink}$ offers no more information than an immediately previous position $\mathcal{V}_{eval}$).

Semantics of the Boolean game. Short name: $Sem^{\mathbb{B}}$

The semantics of an arbitrary $(\mathbb{N}^n \times \mathbb{N}^m)^i \models \varphi$ are also essentially analogous to numeric Lemma 3.20.

However, now the n and m quantify over finite choices: that is, we have to substitute $\overline{x} \in 2^{|Num|}$ with $\overline{e} \in 2^{|\mathbb{B}|}$ and $\overline{y} \in 2^{|Num|}$ with $\overline{s} \in 2^{|\mathbb{B}|}$.

Transitions in the Boolean game. Short name: $\mathcal{T}^{\mathbb{B}}$

Also the analogy with Definition 3.21 happens for the transition, except for, again, the the sink positions. See it in Definition 3.24.

***Definition* 3.24.** *The system can lead us to directly making the value of b false. Let evalExtraReq be the function that 'evaluates the extra requirement with the $\overline{e}$ and $\overline{s}$ valuations', that is, given $\overline{e}$ and $\overline{s}$, evaluates whether the extra requirements is true or not. We define the semantics as follows:*

$$T_{sys}(v,v') \text{ holds when } v = (b,h,\overline{e}) \text{ and:}$$

$$\begin{cases} v' = (b, h :: (\overline{e},\overline{s})) \text{ for some } \overline{s} \in \mathbb{N}^m, & evalExtraReq(\overline{e},\overline{s}) \\ v' = (\neg b, h :: (\overline{e},\overline{s})) \text{ for some } \overline{s} \in \mathbb{N}^m, & \neg evalExtraReq(\overline{e},\overline{s}) \end{cases}$$

Note that, in the $\neg evalExtraReq(\overline{e},\overline{s})$ case, the system must move to a sink state.

Justification of a temporal theorem

The Local Booleanization Theorem (see Subsection 3.2.1) states that for a non-temporal set of formulae written in a decidable $\exists^*\forall^*$ fragment of a logic, a Booleanization exists and it is computable.

This means that if the numeric specification models a game, then so does its Boolean specification. Thus, we could inspect both games and, in case the are equi-realizable, then Booleanization yields a realizability-checking procedure for the game that has resulted from a Booleanization of a non-temporal numeric specification and, in consequence, a realizability-checking procedure for the original numeric specification.

We proved this can indeed be done in Lemma 3.10 (the reason to prove it later is because we have not introduced the necessary graphic notation yet).

However, this is not a solution for the original intention of the thesis, since we want to check equi-realizability of **temporal** games.

To illustrate this, assume the next situation:

1. We have received a a certain numeric LTL specification φ^{Num}.
2. Given φ^{Num}, we have constructed a numeric game $\mathcal{G}^{Num}$.
3. Booleanizing φ^{Num}, we obtain $\varphi^{\mathbb{B}}$, which is the Booleanized version of the specification.
4. Given $\varphi^{\mathbb{B}}$, we have constructed a Boolean game $\mathcal{G}^{\mathbb{B}}$

The question is: are $\mathcal{G}^{Num}$ and $\mathcal{G}^{\mathbb{B}}$ equi-realizable?

The Local Booleanization algorithm we have seen in Subsection 3.1.2 only works using the variables (see Definition 3.1) and configurations (see Definition 3.2); and, using them, produces what we called the *reactions* (see Definition 3.5) and then the equivalent Booleanized specification (whose main contribution is the extra requirement it produces -see Subsubsection 3.1.2).

That means this process is ignoring the *Yesterday* (Subsubsection 2.1.3) or *Next* (see Subsection 2.1.3) operations that literals could have nested: that is, it is ignoring temporality.

So someone could conjecture that for specifications using LTL (see Example 3.14 below) the Booleanization does not work properly.

Example **3.14.** *Let a numeric LTL specification be* $\varphi = \Box x \in \mathbb{E} \wedge y \in \mathbb{S} :: \mathcal{Y}(x < y)$.

Now, the Booleanization algorithm of Subsection 3.1.2 first ignores the $\mathcal{Y}$ *operator and Booleanizes the non-temporal specification* $\varphi_{(no\ \mathcal{Y})} = (x < y)$.

And let the Booleanization of $\varphi_{(no\ \mathcal{Y})}$ *be* $\varphi^{\mathbb{B}}_{(no\ \mathcal{Y})} = \Box(s \wedge \varphi_{extra})$, *where* φ_{extra} *represents the extra requirement.*

The key question is: would it be correct to simply add the temporal operator $\mathcal{Y}$ *to the Booleanized literal* s *in* $\varphi^{\mathbb{B}}_{(no\ \mathcal{Y})}$ *(which is representing the literal* $(x < y)$ *in* $\varphi_{(no\ \mathcal{Y})}$*)? Adding the operator, the resulting formula would be:* $\varphi^{\mathbb{B}} = \Box(\mathcal{Y}(s) \wedge \varphi_{extra})$.

Informally, we can conjecture that the Booleanized temporal specification $\varphi^{\mathbb{B}}$ of Example 3.14 above is correct. We show now an example of the Booleanization, which, by the first theorem should informally hold in Example 3.15 below.

Example **3.15.** *Once again, this example is based in Example 3.6. Consider the next specification:*

Let $\varphi = \Box(\varphi_{Req_0} \wedge \varphi_{Req_1})$, *where:*

$$Req_0 \equiv (x > 1000 \implies y \leqslant x) \wedge \mathcal{Y}(x \leqslant 1000 \implies y > x)$$
$$Req_1 \equiv \mathcal{Y}(y > x)$$

We can see φ *is still unrealizable (the only difference is that the second literal is operated by a Yesterday (see Subsection 2.1.3)).*

Now, the Booleanization will, first, ignore the temporality and raise the following result (we will ignore the extra requirement since it is not temporal):

$$\varphi^{\mathbb{B}} = \square(req_{0_{Bool}} \wedge req_{1_{Bool}} \wedge \varphi_{extra})$$

where $req_{0_{Bool}} \equiv (s_0 \implies s_1) \wedge (\neg(s_0) \implies \neg(s_1))$ *and* $req_{1_{Bool}} \equiv (\neg(s_1))$.

And then add the temporality:

$$\varphi^{\mathbb{B}} = \square(req_{0_{Bool}} \wedge req_{1_{Bool}} \wedge \varphi_{extra})$$

where $req_{0_{Bool}} \equiv (s_0 \implies s_1) \wedge \mathcal{Y}(\neg(s_0) \implies \neg(s_1))$ *and* $req_{1_{Bool}} \equiv \mathcal{Y}(\neg(s_1))$.

This specification is also unrealizable, as the original one, and preserves the original powers of each player.

Therefore, since it seems like a conjecture, we can informally say that, by inspection, if:

1. We are given a numeric LTL specification φ^{Num}.
2. We ignore its temporal operators, obtaining $\varphi^{Num}_{no\ temp}$.
3. From $\varphi^{Num}_{no\ temp}$ we construct an analogous Boolean specification $\varphi^{\mathbb{B}}_{no\ temp}$.
4. We properly add the temporal operators in the Booleanly substituted literals of $\varphi^{Num}_{no\ temp}$, resulting in $\varphi^{\mathbb{B}}$.

Then, $\varphi^{\mathbb{B}}$ is a correct Booleanization of φ^{Num}: that is, $\varphi^{\mathbb{B}}$ preserves the original power of each variable-owner φ^{Num}. So, it is an equi-realizable Boolean LTL specification of the original one.

Thus, going back to the realizability equivalence checking between a numeric LTL game $\mathcal{G}^{Num}$ and its supposedly equivalent Boolean LTL game $\mathcal{G}^{\mathbb{B}}$, the informal statement we made in the paragraph above implies that indeed they are equi-realizable. Since this is the final objective of the thesis, we need to formally proof that this informal statement holds.

To achieve this, the aim is to relate the two games, and that starts by stating a more expressive property, that is expressed in Claim 3.4 below.

Claim **3.4.** *Let φ^{Num} be a certain numeric LTL specification, $\mathcal{G}^{Num}$ be the numeric game given by φ^{Num} and $\mathcal{G}^{\mathbb{B}}$ be the supposedly equivalent Boolean game given by the $\varphi^{\mathbb{B}}$, which is the Booleanized version of φ^{Num}.*

The positions of $\mathcal{G}^{Num}$ (see Definition 3.18) and the positions of $\mathcal{G}^{\mathbb{B}}$ (see Definition 3.1) can be related with relation $\mathcal{R}$ (which is more restrictive than the cross product): a one-to-(potentially)many correspondence, where each Boolean position can correspond to more than one equivalent numeric position.

Proof. We have indeed defined the relation $\mathcal{R}$ (see Definition 3.25) in the next Subsubsection 3.2.2. □

Note that this, in other words, implies that every movement of one game can be mimicked by the other one; that is, that every strategy of one game can be mimicked by the other one. In conclusion, that both games are equivalent in terms of realizability.

The idea is that for any $\mathcal{G}^{Num}$ and its supposedly equivalent $\mathcal{G}^{\mathbb{B}}$, and given the position of one, we can guess the position of the other one because of this relation.

This can be more easily understood attending to Example 3.16 below.

Example **3.16.** *We can depict an arbitrary $\mathcal{G}^{Num}$ and an arbitrary $\mathcal{G}^{\mathbb{B}}$ and see how they are related.*

To do so, we will use Figure 3.4, in which:

- *The bottom arena represents the Numeric one (see Definition 3.16), while the top arena is the Boolean one (see Definition 3.1).*
- *There are some extra Init positions that represent an initialization state of a new timestep, that is, their evaluation is just true.*
- *The diamond means the position is initial, the round means the position belongs to the environment, the box mean the position belongs to the system, the triangle means the position is evaluable (all these position types are described in Definition 3.18 and Definition 3.1).*
- *The white surface cover (which only happens in triangle positions and in box positions of $\mathcal{G}^{\mathbb{B}}$) means the state is violating the specification.*

- *$\mathcal{Y}^n$ means the maximum length of yesterdays (i.e. the so-called temporal order).*
- *There are some key boxes $\boxminus$ linked, like: $< \boxminus\mathcal{Y}^n, \boxminus\mathcal{Y}^{n-1}, .., \boxminus\mathcal{Y}, \boxminus_{Now} >$. Each of them represents the valuations of both players made in each timestep until the allowed temporal length. In all of them, a $\times$ symbol inside the upper minibox means the valuations for the environment have been performed, while the down minibox means the valuations for the system have been performed (after the ones of the environment). Note that when they are blank, it means no valuation has been made yet, because (1) the temporality of the minibox has not been reached yet (i.e. if falls off the trace) or (2) it is the current box, denoted by $\boxminus_{Now}$ and we are still in an init position or in a environment position.*
- *Positions are not one-to-one related, but one-to-many. One position in $\mathcal{G}^{\mathbb{B}}$ can be related with many positions of $\mathcal{G}^{Num}$. To represent this infinity in transitions of $\mathcal{G}^{Num}$, a double arrow $\Longrightarrow$ is used. On the contrary, to represent multiple but finite transitions, a cut by the finite-number n of transitions $/^n$ is used, as it is the norm in classical digital electronics' notation.*

Concretely, in Figure 3.4, the represented situation is one in which the temporal depth is 2 and we are currently playing a timestep $t \geqslant 2$, since note that both the previous timestep (represented with a $\boxminus\mathcal{Y}$) and the one before (represented with a $\boxminus\mathcal{Y}\mathcal{Y}$) are filled with valuations (represented with a $\times$). After the new two valuations on the right top, the specification has resulted true, since, otherwise, the triangles of the end would be double-marked.

Thus, we can see in Figure 3.4 how each numeric-Boolean position is related by the Local Theorem (see Subsection 3.2.1). However, for the evaluable positions, it does not suffice to use that Local Theorem: indeed some kind of relation is needed.

Remember that the positions (note, mainly, the evaluable positions) also store the information of whether there has been any violation of the specification (i.e. a $\bot$ in an evaluable position of whether $\mathcal{G}^{Num}$ or $\mathcal{G}^{\mathbb{B}}$, or the visit of a Sink state in $\mathcal{G}^{\mathbb{B}}$) in a bit called *b_histo* (see Lemma 3.3).

Having this bit is essential to ensure equi-realizability between both games, concretely to ensure the violation results are not lost over the infinite game. In other words: it could be the case (see Example 3.17 below) that in a given timestep the specification has been

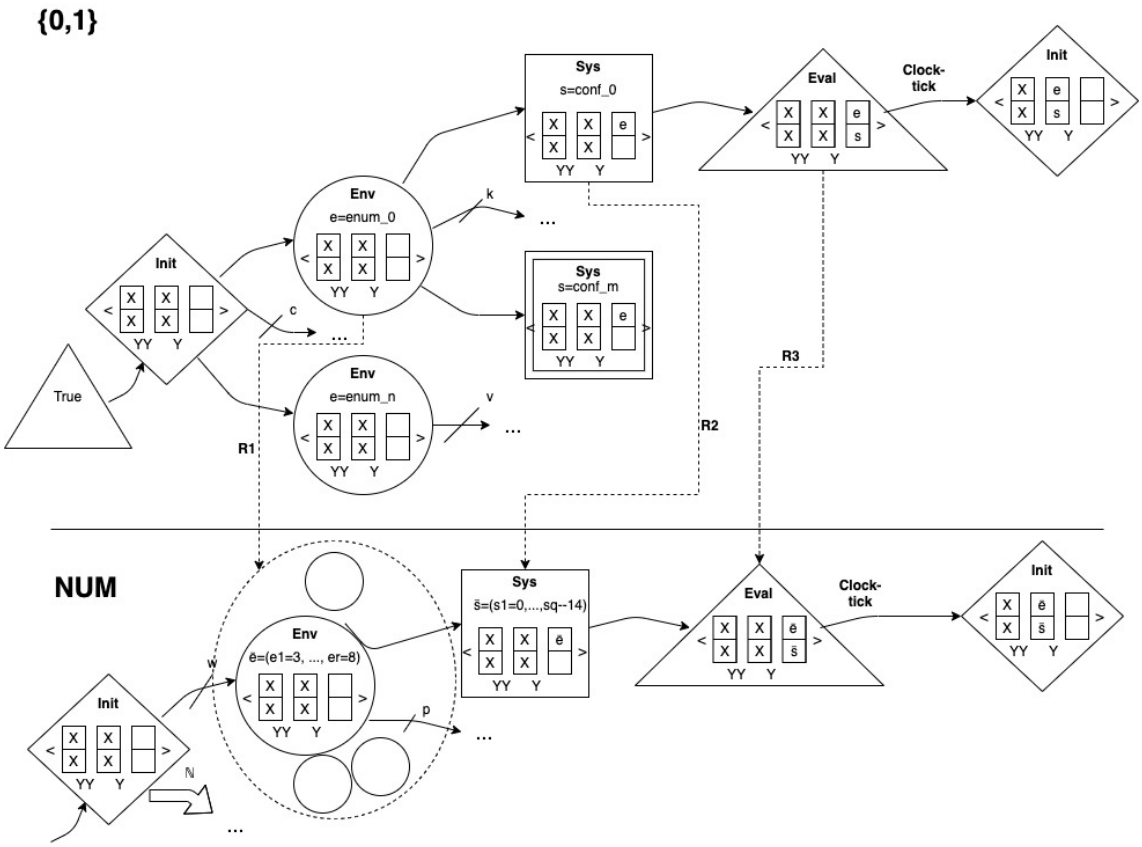

Figure 3.4: An arbitrary numeric game and its equivalent Boolean game, and the relation between their positions.

Source: Own.

violated (in both games), but, due to temporal reasons, that violation is forgotten and the rest of the infinite games preserve equi-realizability. Thus, considering those games equi-realizable would be a wrong result.

Example **3.17** (A history bit is needed to ensure equi-realizability)**.** *A simple example that shows the need of a history bit b_histo is an arbitrary temporal game without temporal operators (i.e. the temporal depth t_k -see Definition 3.13- of the specification is* 0*), that has violated an specification and never violates it again.*

Since $t_k = 0$, then the (wrong) result would be realizable in both games. This is illustrated in Figure 3.5 below.

We can see in Figure 3.5 above that in timestep 0 *there has been a specification violation that has not been stored, and, thus, the realizabiliy checking of the games has returned a wrong realizable answer. Also note that the evaluable positions are not specified separa-*

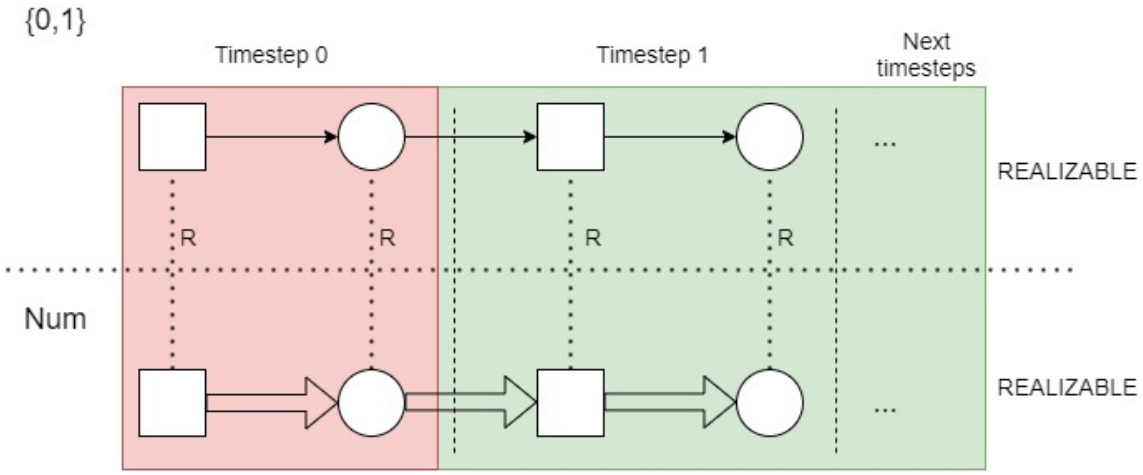

Figure 3.5: A simplified graph, where the squares denote environment positions and the round denote system positions (that is, it does not show the needed information that each positions carries -see Definition 3.18) and where red colour denotes unrealizability, contrary to the green colour. Also, double arrows mean potentially infinite transitions.

Source: Own.

tedly for simplificity reasons, and are merged with the environment positions.

Contrary to Figure X, b_histo would make the case that if there is a violation, it will always be marked for the rest of the game, since, remember: it is conjuncted with the current evaluation: game_eval = b_histo ∧ current_eval. We can see that in Figure 3.6 below.

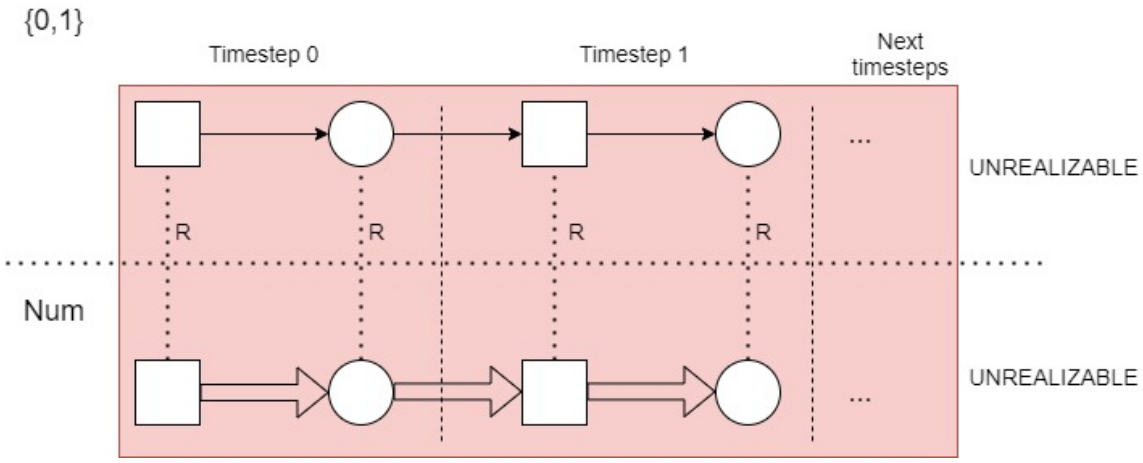

Figure 3.6: A version of Figure 3.5 where the history bit has been applied, producing a different realizability result.

Source: Own.

We can see in Figure 3.6 above that, thanks to the history bit, every section of the trace once there has been a violation is unrealizable (and, therefore the games themselves).

Coming back to the topic, we will retake Example 3.16, we can see, some positions of the Boolean arena are related with positions of the numeric one. To see an actual case where this happens, we can check Example 3.18 below.

Example **3.18.** *We will use the same input as in Example 3.6:*

$$Req_0 \equiv (x > 1000 \implies y \leqslant x) \wedge (x \leqslant 1000 \implies y > x)$$

Where $Sys_0 \equiv (x > 1000)$, $Sys_1 \equiv (y \leqslant x)$, $\neg Sys_0 \equiv (x \leqslant 1000)$ *and* $\neg Sys_1 \equiv (y > x)$.

Let us consider now that, a refined Booleanization has given us the following extra requirement (see Definition 3.10):

$$\begin{aligned} e = e_1 &\implies (Sys_1 \wedge \neg Sys_0) \vee (\neg Sys_1 \wedge \neg Sys_0) \\ &\wedge \\ e = e_2 &\implies (Sys_1 \wedge Sys_0) \vee (\neg Sys_1 \wedge Sys_0) \end{aligned}$$

We will show in Figure 3.7 a frame of both the Boolean and the numeric arena, and the relation between some of its positions.

We know (and can see in Example 3.18 above) that non-evaluable position's equivalence trivially hold because of the Booleanization of non-temporal requirements (see Subsection 3.1.1).

Concretely, we are stating that a relation $\mathcal{R}$ [2] is holding between them: if a Boolean evaluable position's verdict is not $\perp$ (i.e. it is not a Bad position -see Definition 3.17), then its related numeric position's verdict is also not $\perp$ and, moreover, each of them can move to another evaluable position (after n steps) which is related with the other one. A summary of this idea can be seen in Figure 3.8.

However, evaluable positions $\mathcal{V}_{eval}$ need some extra relation between them, because they refer to LTL specifications.

[2] The relation is close to the *bisimulation* concept, but it is not properly a bisimulation.

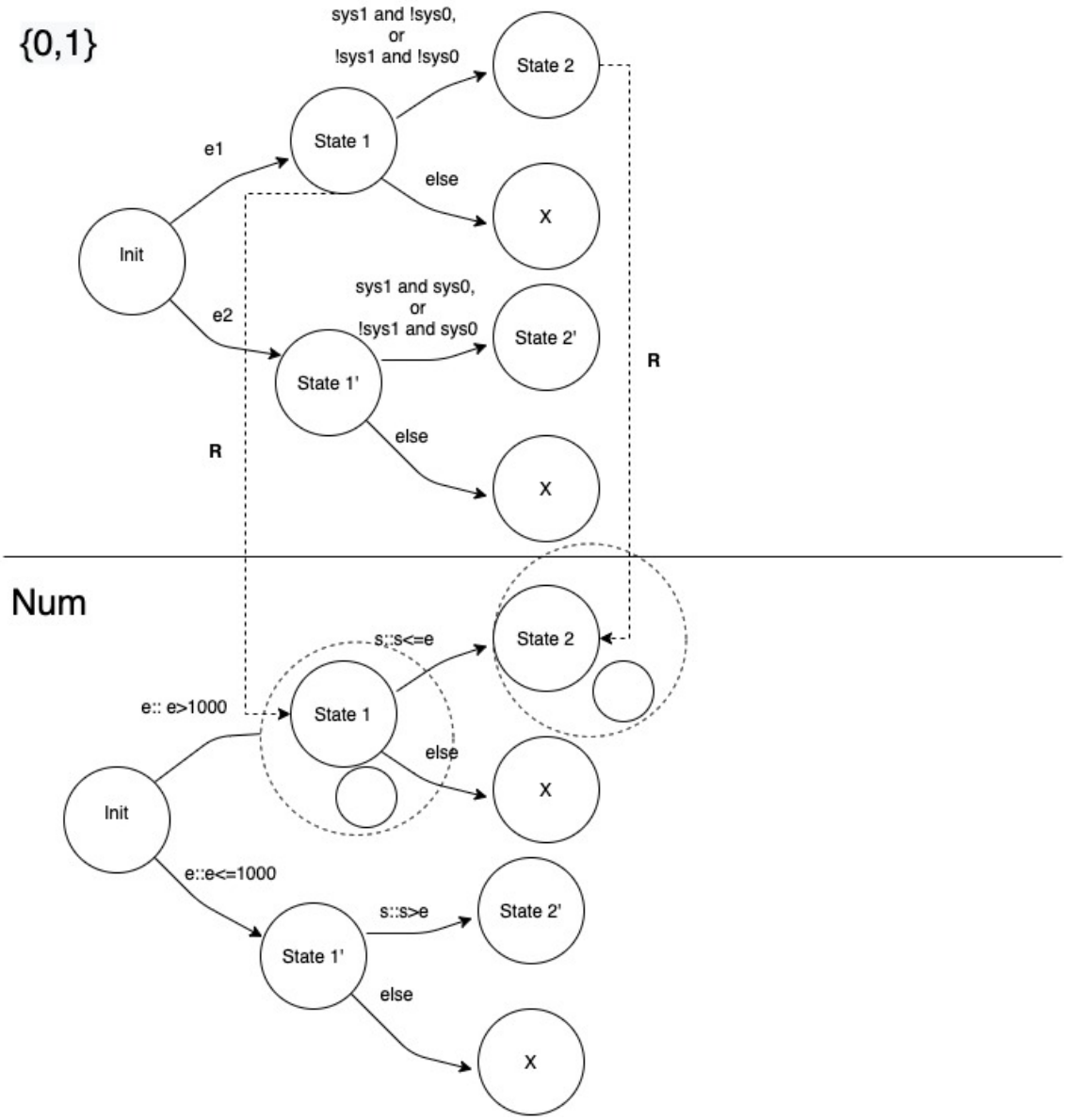

Figure 3.7: A Booleanized and numeric arenas' frames, and the relationship between some states.

Source: Own.

The key relation $\mathcal{R}$ over the positions of both games

We have stated in Subsubsection 3.2.2 above that, when talking about temporal specification φ_{Num}, stating that the evaluable positions of a numeric game given by φ_{Num}, and its Boolean equivalent game (given by $\varphi_{\mathbb{B}}$) are equivalent does not follow directly from the *Local Booleanization theorem* in 3.2.1.

That is: due to its temporal nature, the evaluable positions $\mathcal{V}^{Num}$ of a numeric game and the

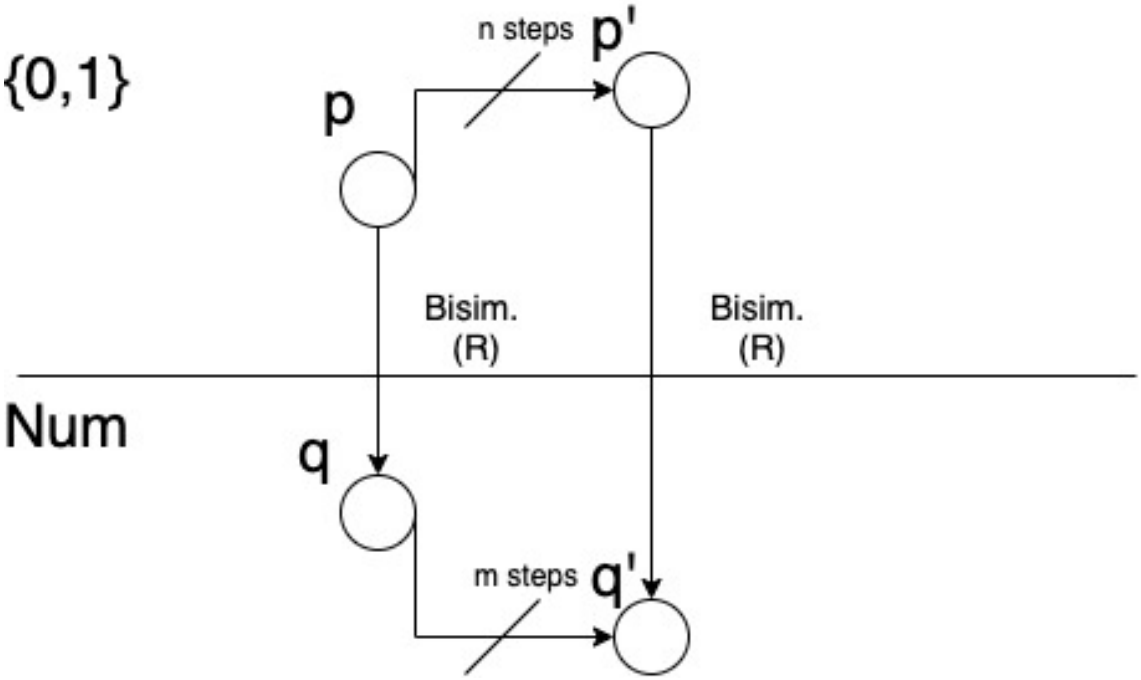

Figure 3.8: An arbitrary p position of the Boolean game is related with an arbitrary q position of the numeric game; and their post-positions are also related.

Source: Own.

ones $\mathcal{V}^{\mathbb{B}}$ of its supposedly equivalent a Boolean game, are not trivially *'equi-verdictable'* (i.e. they they have the same verdict -see Definition 3.17- result, but it is not easy to prove this).

The reason for this is that the both the $\mathcal{V}^{Num}$ and the $\mathcal{V}^{\mathbb{B}}$ have to keep track of the valuations of the variables made in the previous timesteps (see Definition 3.18), until a temporal length k given by the maximum number of nested Yesterday operators (see Subsubsection 2.1.3).

Thus, the key definition here is this relation $\mathcal{R}$ between an arbitrary $\mathcal{V}^{Num}$ and its $\mathcal{V}^{\mathbb{B}}$, which is given in Definition 3.25 below.

***Definition* 3.25** (A relation $\mathcal{R}$ between positions)**.** *Let $\mathcal{P}^{num}$ be the set of positions in an arbitrary numeric game, and $\mathcal{P}^{\mathbb{B}}$ be the set of positions of its Boolean game.*

Now, given two arbitrary positions $\mathcal{V}^{Num} \in \mathcal{P}^{Num}$ and $\mathcal{V}^{\mathbb{B}} \in \mathcal{P}^{\mathbb{B}}$, the relation function $\mathcal{R} \subseteq \mathcal{P}^{Num} \times \mathcal{P}^{\mathbb{B}}$ states that under certain conditions (see below) then they have a one-to-one correspondence.

When $\mathcal{V}^{Num}$ and $\mathcal{V}^{\mathbb{B}}$ hold $\mathcal{R}$, it is denoted as follows:

$$\mathcal{V}^{Num}\ \mathcal{R}\ \mathcal{V}^{\mathbb{B}}$$

So that $\mathcal{R}$ holds, the relation needs two pre-requisites on the arbitrary numeric position $\mathcal{V}_{Num}$ and the Boolean position $\mathcal{V}_{\mathbb{B}}$. That is, $\mathcal{R}$ holds whenever:

1. *Both $\mathcal{V}^{Num}$ and $\mathcal{V}^{\mathbb{B}}$ belong to the same timestep of the trace: in other words, if their timestep is not the same, then $\mathcal{R}$ cannot hold between them. Formally:*

$$\textit{If } (\mathcal{V}^{Num^{t_i}} \wedge \mathcal{V}^{\mathbb{B}^{t_j}} \wedge i \neq j) \textit{ then } (\mathcal{V}^{Num^{t_i}} \mathcal{R} \mathcal{V}^{\mathbb{B}^{t_j}} \implies \bot)$$

2. *$\mathcal{V}^{\mathbb{B}}$ is not a Sink position (see Definition 3.1). Formally:*

$$\mathcal{V}^{\mathbb{B}} \notin \textit{Sinks}$$

   Where Sinks is the set of Sink positions.

3. *$\mathcal{V}^{Num}$ is an evaluable position if and only if so is $\mathcal{V}^{\mathbb{B}}$. Formally:*

$$\mathcal{V}^{Num} \in \mathcal{P}^{Num}_{Eval} \textit{ iff } \mathcal{V}^{\mathbb{B}} \in \mathcal{P}^{\mathbb{B}}_{Eval}$$

4. *If both $\mathcal{V}^{Num}$ and $\mathcal{V}^{\mathbb{B}}$ are evaluable positions, then $\mathcal{V}^{Num}$ conforms (i.e. does not violate) the original numeric specification φ^{Num} if and only if $\mathcal{V}^{\mathbb{B}}$ conforms its Booleanized specification $\varphi^{\mathbb{B}}$. Formally:*

$$\textit{If } (\mathcal{V}^{Num}_{eval} \wedge \mathcal{V}^{\mathbb{B}}_{eval}) \textit{ then } (\mathcal{V}^{Num}_{eval} \models \varphi^{Num} \leftrightarrow \mathcal{V}^{\mathbb{B}}_{eval} \models \varphi^{\mathbb{B}})$$

Note that, properly stated, the correspondence given by $\mathcal{R}$ is not a one-to-one correspondence, but rather a one-to-(potentially)many correspondece: since, a single Boolean position is related with an (potentially infinite) set of numeric positions. The one-to-one correspondence would be a particular case, where the number of positions in the numeric game is finite. An example of the one-to-many correspondence can be seen in Figure 3.4.

Once the conditions are fulfilled, the properties that $\mathcal{R}$ holds are listed in Lemma 3.4 below.

Lemma **3.4** (Properties of $\mathcal{R}$)**.** *Let, as in Definition 3.25, $\mathcal{P}^{num}$ be the set of positions in an arbitrary numeric game, and $\mathcal{P}^{\mathbb{B}}$ be the set of positions of its Boolean game.*

Now, given arbitrary numeric position $\mathcal{V}^{Num}$ and the Boolean position $\mathcal{V}^{\mathbb{B}}$, the relation $\mathcal{R}$ over them holds has properties:

1. If there is a transition from a position $\mathcal{V}^{Num}$ (call it p) to another position $\mathcal{V}'^{Num}$ (call it p'), then there is a transition from $\mathcal{V}^{\mathbb{B}}$ (call it q) to another position $\mathcal{V}'^{\mathbb{B}}$ (call it q'), and p' and q' are related . And the same the other way. Formally:

$$\begin{aligned} &\textit{If } (p' \in \mathcal{P}^{Num}) \textit{ and } ((p,p') \in \mathcal{T}^{Num}) \textit{ then} \\ &(\exists q' :: (q' \notin \textit{Sink}) \wedge (q' \in \mathcal{P}^{\mathbb{B}}) \wedge (q,q') \in \mathcal{T}^{\mathbb{B}}) \\ &\wedge (p' \mathcal{R} q')) \end{aligned}$$

   This property is true because of the Local Booleanization Theorem (see Subsection 3.2.1), that ensures a relation between numeric positions and Boolean positions in a non-temporal game.

2. The same of the point above, but on the other way. Formally:

$$\begin{aligned} &\textit{If } (q' \in \mathcal{P}^{\mathbb{B}} \wedge q' \notin \textit{Sink}) \textit{ and } ((q,q') \in \mathcal{T}^{\mathbb{B}}) \textit{ then} \\ &(\exists p' :: (p' \in \mathcal{P}^{Num}) \wedge (p,p') \in \mathcal{T}^{Num}) \\ &\wedge (p' \mathcal{R} q')) \end{aligned}$$

   The proof for it is analogous to the previous one.

3. For every $\mathcal{V}^{Num}$ (call it p), there is a unique $\mathcal{V}^{\mathbb{B}}$ (call it q) with which it is related. Formally:

$$\textit{If } \forall p :: p \in \mathcal{P}^{Num} \textit{ then } (\exists q :: q \in \mathcal{P}^{\mathbb{B}} \wedge p \mathcal{R} q) \wedge (\nexists q' :: q' \in \mathcal{P}^{\mathbb{B}} \wedge p \mathcal{R} q' \wedge q \neq q')$$

   The proof for it is given in Lemma 3.5.

4. For every $\mathcal{V}^{\mathbb{B}}$ (call it p) there is at leas one $\mathcal{V}^{Num}$ (call it q) with which it is related. Formally:

$$\textit{If } \forall q :: q \in \mathcal{P}^{\mathbb{B}} \wedge q \notin \textit{Sink then} (\exists p :: p \in \mathcal{P}^{Num} \wedge p \mathcal{R} q)$$

   The proof is also given in Lemma 3.5.

We will see later that these properties of Lemma 3.4 above imply the Theorem 3.2.

But, before that, we still have to prove properties (3) and (4) of Lemma 3.4, which has been done using the structural induction in Lemma 3.5 below.

Lemma **3.5** ($\mathcal{R}$ yields equi-verdictability of evaluable positions)**.** *Every Boolean evaluable position (see Definition 3.18) is related by the relation $\mathcal{R}$ with some numeric evaluable position of the same timestep (see Definition 3.11); and vice versa (i.e. both positions are equi-verdictable).*

Proof. Let us use $\boxminus$ to denote whether a player has made a valuation of its variable. When it is empty, it means no valuation has been done; then $\boxminus_{\mathbb{E}}$ means the environment has chosen its variables and $\boxminus_{\mathbb{ES}}$ means that both the environment and then the system have chosen their variables.

*The reason to use $\boxminus$ is that it is similar, for instance, to the $\boxminus$ used in Figure 3.4.

Now, we have a couple of facts:

- Fact 1: We know that if a literal falling off the trace (see Definition 3.14), has the verdict $\top$. Formally:

$$\sigma \vDash \mathcal{Y}(\varphi) \textit{ and } |\sigma| \leqslant 1 \textit{ then } \sigma \vDash \mathcal{Y}(\varphi) \textit{ holds}$$

  Then, if the current timestep k has not reached the temporal depth n, that means both the Boolean valuations ($\{\boxminus^{\mathbb{B}}\}$) and the numeric ones ($\{\boxminus^{Num}\}$) are empty. Then, since it holds that both a Boolean evaluable position (which uses $\{\boxminus^{\mathbb{B}}\}$) and a numeric evaluable position (which uses $\{\boxminus^{Num}\}$) have both the *true* result, then they have the same one -i.e. the relation $\mathcal{R}$ holds. Formally:

$$\forall\, \{\boxminus_{\mathcal{Y}^k}\} :: (k < n) \implies (pos^{\mathbb{B}}(\{\boxminus^{\mathbb{B}}\}) \leftrightarrow pos^{Num}(\{\boxminus^{Num}\}))$$

- Fact 2: By the Local Booleanization Theorem (see Subsection 3.2.1), we also know that:
  - The non-evaluable positions of both are related with $\mathcal{R}$ trivially because no evaluation change is made in them.
  - If winning, the two evaluable positions are related because of Theorem 3.1.

Therefore, we can inductively (by structural induction over the arenas) state that the equi-realizability on the previous timesteps has held whether because of Fact 1 or because of Fact 2; and so will do the current timestep.

We can see a picture of this structural process in Figure 3.9 below.

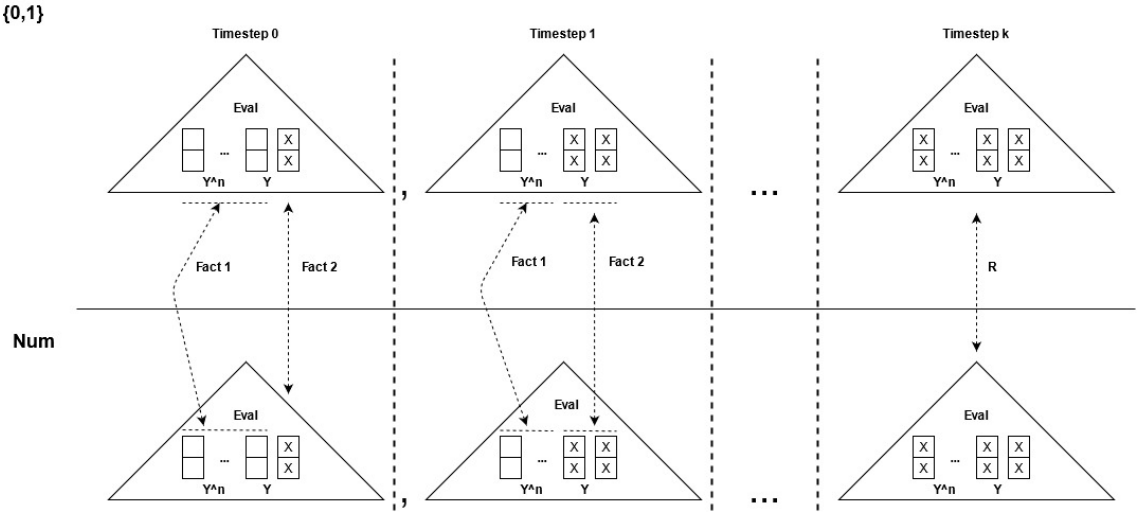

Figure 3.9: A visual representation of the structural proof of Lemma 3.5, where k is greater or equal than the temporal depth (see Definition 3.13).

Source: Own.

□

Remember that, due to the verdict control bit (see b_histo in Lemma 3.3), in case the specification is violated in any moment of any the game (whether (1) because of a $\perp$ verdict in any game's evaluable position, or (2) because of visiting a Sink position -see Definition 3.1- in the Boolean game), then the $\perp$ result will remain forever, even if the rest of the game is again equi-realizable.

Lemma 3.5 above can also be formally explained in terms of logic: we can construct a syntactic tree for both the numeric specification and its Booleanized specification (in the same way as in Example 3.12) and see how the subformulae in them (indeed, the literals in them) are related. To do so, we will again use *Fact 1* and *Fact 2* from Lemma 3.5 above, which has been shown in Lemma 3.6 below.

Lemma **3.6** (Lemma 3.5 vis: using a tree)**.** *Let us consider an arbitrary numeric LTL specification and its Booleanized LTL specification. We can construct a syntactic tree for each specification for each, as we can see Figure 3.10 below.*

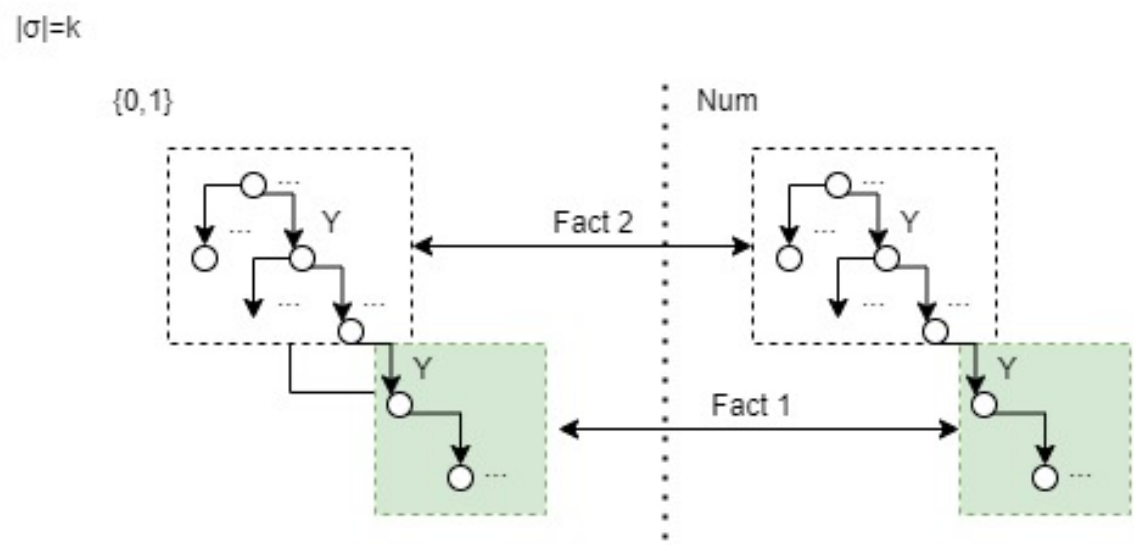

Figure 3.10: A visual representation of proof of Lemma 3.6, using the same construction of Example 3.12. Here, $k \geqslant 0$ is an arbitrary trace length (see Definition 3.12), while '$\cdots$' denotes that a set of logic operators (whether temporal or not) have been used or not.

Source: Own.

Now, using Fact 1 and Fact 2 from Lemma 3.5, we can graphically see that whether because of Fact 1 or Fact 2, every subformulae of each specification is related, since every literal is related.

Therefore, $\mathcal{R}$ yields equi-verdictability of evaluable positions.

Note that, in both Lemma 3.5 and Lemma 3.6 above, we are always considering winning games (for the system), because we want to avoid Sink positions (see Definition 3.1) of the Boolean game, since there is not Sink equivalent in the numeric game. This is a correct reasoning, as winning is dual to losing: so ensuring winning equivalence ensures realizability equivalence (see Claim 3.6, whose result is the same as Claim 3.5 below).

Claim **3.5.** *As a corollary to the previous Lemma 3.5, we know that if a Sink position is reached in the Boolean game, then the result of the evaluable position of the numeric game has been UNSAT (i.e. the verdict has been $\perp$). Formally:*

$$\exists \mathcal{V}^{\mathbb{B}} \text{ such that } \mathcal{V}^{\mathbb{B}} \in \mathit{Sinks} \leftrightarrow \exists \mathcal{V}^{Num} \text{ such that } \mathcal{V}^{Num} \in \mathcal{B}_{Num}$$

where Sinks is again the set of Sink positions and $\mathcal{B}_{Num}$ is the set of bad positions (see Definition 3.17) in the numeric game.

This is so, because the system cannot choose an invalid reaction in $Game_{\mathbb{B}}$ (by the Reaction Existence Lemma 3.1, we know $\forall \bar{x}. \bigvee_{r \in VR} r$): it would the same as choosing a valuation that would make the result Unsat the $Game_{Num}$, which would not make game-sense.

Game mimicking theorem

We will extend the relation $\mathcal{R}$ to plays and strategies.

We see, by Lemma 3.7 below, that under the conditions detailed in Definition 3.25, a winning play in the numeric game is related by $\mathcal{R}$ to a play in the Boolean game.

Lemma **3.7.** *Let $\pi \in Plays(\mathcal{G}^{Num})$ and let π' be the set of plays such that $\pi^i \mathcal{R} \pi'^i$. It is easy to see (by 2, 3 and 4 in Lemma 3.4) that there is a unique π' and that π' is a legal play of the Boolean game $\mathcal{G}^{\mathbb{B}}$.*

Moreover, for every j such that $\pi^j \in \mathcal{B}^{Num}$ (see Definition 3.17) if and only if $\pi^j \in \mathcal{B}^{\mathbb{B}}$: that is, π is winning for the system in one game if and only if, it is winning for the other game. That means plays of both games are related. Formally:

$$\pi \sim_{\mathcal{R}} \pi'$$

Now, we can set the equivalency between numeric and Boolean strategies in Lemma 3.8 below.

Lemma **3.8.** *A strategy ρ in $\mathcal{G}^{Num}$ is related with another strategy ρ' in $\mathcal{G}^{\mathbb{B}}$. Formally:*

$$winning(\rho^{Num}) \leftrightarrow winning(\rho^{\mathbb{B}})$$

Proof. Let ρ be a strategy for the system in $\mathcal{G}^{Num}$. We now build ρ' in $\mathcal{G}^{Num}$ such that for every play $\pi \in Plays(\rho)$ there is a play $\pi' \in Plays(\rho')$ with $\pi \sim_{\mathcal{R}} \pi'$ and vice versa.

To do so, we build ρ' from ρ as follows: let P_s denote the positions of the system, let $q \in P^{\mathbb{B}}_{System}$ be arbitrary and $p \in P^{Numeric}_{System}$ be its unique (by (3) in Lemma 3.4) $p \mathcal{R} q$.

Let $p' = P(\rho)$ and let q' be (by (4) in Lemma 3.4) the unique $q \in P^{\mathbb{B}}_s$ such that $p' \mathcal{R} q'$. Then, we let $p'(q) = q'$.

We can see by induction that if π is a play played according to ρ and $\pi \sim_{\mathcal{R}} \pi'$ then pi' is

played according to ρ'. Therefore, formally:

$$\pi \in Plays(\rho) \leftrightarrow \pi' \in Plays(\rho')$$

Even futher: ρ is winning is ρ' is winning. □

And we can check it for the other direction (i.e. Boolean strategy to numeric strategy) in Lemma 3.9 below.

Lemma **3.9.** *A strategy ρ in $\mathcal{G}^{\mathbb{B}}$ is related with another strategy ρ' in $\mathcal{G}^{Num}$. Formally:*

Proof. For the other direction, we assume a ρ' such that a position of any play of ρ' never goes to a sink. Now, we will build a ρ.

Note that if the System can win $\mathcal{G}^{\mathbb{B}}$ then there is a ρ' that never visits a sink.

The construction is analogous to the previous Lemma 3.8. □

We can now instantiate the foundations of infinite games seen in Subsection 2.2.1, the Local Theorem in Subsubsection 3.2.1 and the properties of $\mathcal{R}$ stated in Lemma 3.4 to raise a new Temporal Theorem, which can be seen in Theorem 3.2 below.

Theorem **3.2.** *The system wins the numeric game $\mathcal{G}^{Num}$ if and only if the system wins the game $\mathcal{G}^{\mathbb{B}}$ produced by its Booleanized specifications. Formally:*

$$System\ wins\ (\mathcal{G}^{Num}) \leftrightarrow System\ wins\ (\mathcal{G}^{\mathbb{B}})$$

Therefore, since both games are realizable at the same time, they are also unrealizable at the same time. Thus, they are equi-realizable. This is a corollary of Theorem 3.2 above, which can be summarized in Claim 3.6 below.

Claim **3.6.** *Since System wins $(\mathcal{G}^{Num})$ if and only if System wins $\mathcal{G}^{\mathbb{B}}$ (by Theorem 3.2), then: $\mathcal{G}^{Num}$ is realizable if and only if $\mathcal{G}^{\mathbb{B}}$ is realizable. Formally:*

$$System\ wins\ (\mathcal{G}^{Num}) \leftrightarrow System\ wins\ (\mathcal{G}^{\mathbb{B}}) \implies realizable(\mathcal{G}^{Num}) \leftrightarrow realizable(\mathcal{G}^{\mathbb{B}})$$

The equi-realizability between a temporal numeric game (indeed any temporal game of a $\exists * \forall *$ fragment of a logic) and its Booleanized game is the main result of the thesis.

As a corollary of this result, we can also state that the equi-realizability of non-temporal games hold (see Lemma 3.10 below).

Lemma **3.10** (A corollary: Non-temporal games also hold equi-realizabiliy)**.** *Let* φ^{Num} *and* $\varphi^{\mathbb{B}}$ *be non-temporal specifications, where* $\varphi^{\mathbb{B}}$ *is the Booleanization of* φ^{Num}*. Also, let* $\mathcal{G}^{Num}$ *and* $\mathcal{G}^{\mathbb{B}}$ *be the games constructed by* φ^{Num} *and* $\varphi^{\mathbb{B}}$ *respectively. Then,* $\mathcal{G}^{Num}$ *and* $\mathcal{G}^{\mathbb{B}}$ *are equi-realizable. Formally:*

$$realizable(\mathcal{G}^{Num}) \leftrightarrow realizable(\mathcal{G}^{\mathbb{B}})$$

Proof. It can be derived both from (1) directly the Local Booleanization Theorem (see Subsection 3.2.1) or (2) as a consequence of Lemma 3.5, where the relation to define is a weaker one. □

Lemma 3.10 above means Booleanization yields a realizability-checking procedure for the game that has resulted from a Booleanization of a non-temporal numeric specification and, in consequence, a realizability-checking procedure for the original numeric specification. As mentioned, we can see this result is subsumed by Theorem 3.2.

Interpretation of the Temporal Theorem

By way of recapitulation, now that Theorem 3.2 has been proved, we can see it implies certain facts about both the numeric $\mathcal{G}^{Num}$ and the Boolean game $\mathcal{G}^{\mathbb{B}}$.

To begin with, in both $\mathcal{G}^{Num}$ and $\mathcal{G}^{\mathbb{B}}$, a play is a sequence of π_i, where the environment plays first and then the system plays: $\pi = \pi_{e_1}, \pi_{s_1}, \pi_{e_2}, \pi_{s_2}, ...$ (taking the evaluable positions -see Definition 3.16- for granted).

Also in both games, a game is its corresponding arena (see Definition 3.16 and Definition 3.1) together with the set of winning plays $Win \subseteq Plays(\mathcal{A})$, but the *Win* is different. Then:

- In $\mathcal{G}^{\mathbb{B}}$: *Win* are the set of plays in which it has not been reached (1) a position that violates the extra requirement, nor (2) an evaluable position (after i.e. both players have played) that has the $\perp$ verdict. The number of bad (and good) positions is finite.
- In $\mathcal{G}^{Num}$: *Win* are 'only' the set of plays in which it has not been reached an evaluable position that has the $\perp$ verdict. The number of bad (and good) positions is infinite.

On the other hand, by the relation $\mathcal{R}$ (see Definition 3.25), Theorem 3.2 and Claim 3.6, we know that, on each timestep:

- Every set of (potentially infinite) movements of the environment player in $\mathcal{G}^{Num}$, can be mimicked by the environment player in $\mathcal{G}^{\mathbb{B}}$. And vice versa.
- Every set of (potentially infinite) movement of the system player in $\mathcal{G}^{Num}$ (determined by the previous play of the environment), can be mimicked by the system player in $\mathcal{G}^{\mathbb{B}}$ (also determined by the previous play of the environment). And vice versa
- Every satisfiability evaluation of valuations (i.e. verdicts) of both players will (1) give the same result or (2) give an $\perp$ verdict in the numeric game if an extra-requirement-violating position has been reached in the $\mathcal{G}^{\mathbb{B}}$: that is, maybe in $\mathcal{G}^{\mathbb{B}}$ the evaluable position has been satisfied, but the extra-requirement was violated. This last fact is like that because the Boolean system player choosing an extra-requirement-violating is as equivalent to the numeric system player choosing a non-satisfying assignment of its valuations.

Therefore, a winning (positional) strategy $\rho(w,v) = v'$ for the system consists in, having the valuation of the environment, choosing its own valuations so that an $\perp$ verdict is not reached (nor an extra-requirement-violating position in $\mathcal{G}^{\mathbb{B}}$). A strategy for the environment is exactly the opposite.

And we also know that if play can be mimicked from one game to the other (i.e have the same winner); then, so can be done with the strategies (see Subsubsection 3.2.2). Therefore, using translations $\mathbb{B} \to Num$ and $Num \to \mathbb{B}$, we can derive strategies from plays. Let ρ mean and σ mean *play strategy* in:

$$\rho^{\mathbb{B}} \to^{toNum} \rho^{Num} \text{ and } \rho^{Num} \to^{toBool} \rho^{B}$$
$$\pi^{\mathbb{B}} \in Plays(\rho) \implies toNum(\pi) \in Plays(toNum(\rho)) \text{ and vice versa}$$

In both games, the good positions is all the environment-belonging positions and a subset of the system-belonging positions and a subset of the evaluable positions. We also know that both good and bad positions are disjoint: $Good = \mathcal{S} \backslash \mathcal{B}$.

We also can define some properties of winning regions (call it $\mathcal{W}$) on both games in Lemma 3.11 below.

Lemma **3.11.** *A $\mathcal{W}$ in whichever $\mathcal{G}^{Num}$ or $\mathcal{G}^{\mathbb{B}}$ holds these properties:*

- *Is a subset of the $\mathcal{S}$ positions. Formally:*

$$\mathcal{W} \subseteq \mathcal{S}$$

- *Initial positions belong to the $\mathcal{W}$. Formally:*

$$Init \subseteq \mathcal{W}$$

- *There exists a movement from the winning region to another winning-region-belonging position: that is, the intersection between the movement and the winning region is not empty. Formally:*

$$w \in \mathcal{W} \implies (\exists p.\ Post(w) = p) \wedge (p \in \mathcal{W})$$

- *A union of $\mathcal{W}$ is another $\mathcal{W}$: that is, the WRs are a closed group under the $\cup$ operation.*

In summary

To sum up, the conclusions of this third chapter are two:

- For every decidable $\exists^*\forall^*$ fragment of a logic (even if it is LTL), a correct Booleanization exists and it is computable in $O(2^{2^{lit}})$. This is detailed in Subsection 3.2.1.
- A temporal game of a decidable $\exists^*\forall^*$ fragment of a logic is realizable if and only if its Booleanized game is realizable. As a consequence: $\mathcal{G}^{Num}$ is realizable if and only if its Booleanized $\mathcal{G}^{\mathbb{B}}$ is realizable: therefore, a realizability checking procedure exists for LTL-formulae-based environment-system numeric games. This is detailed in Subsection 3.2.2.

4. CHAPTER

Conclusions

We have researched a realizability problem for numeric LTL formulae and it has been discovered that its solution is not trivial, and cannot be solved with simple strategies like queries to a solver or a bitwise approximation. To solve this problem we have designed a method that (1) not only Booleanizes the predicates, but that also (2) preserves the dependencies between the original variables, as well as (3) decides which variable-owner or player (if requirements model a game) is the owner of each substitution fresh Boolean variable.

The main contribution of this thesis is a theorem which states that a correct Booleanisation exists for all numeric theories with a decidable $\exists^*\forall^*$ fragment. Then, we have a realizability checking procedure for numeric-written safety two-player turn-based LTL games (and, therefore: also for their synthesis). In the practise, this realizability checking can be done using tools for Boolean requirements such as AbsSynthe.

Last but no least, we have proposed and implemented in Ocaml an algorithm that performs the Booleanization.

4.1. Future Work

As for the future work, we outline the following lines:

- Generalization of the brute force algorithm for more theories, such as the theory of complex numbers.

- Enhancement of numeric algorithms using constant quantifier elimination methods.
- Theory mixing incorporation in the Booleanization methods.
- Temporal expresiveness extensions, for instance, with temporal comparisons in some situations.
- Application of the Booleanization in other areas such as Numeric Transition Systems.

4.2. Two parallel researches

In parallel, we worked on two research lines related to Booleanization.

4.2.1. Complexity analysis and heuristics

The first one is the in-depth analysis of the complexity of the Brute force Booleanization algorithm (see Algorithm 1), producing interesting lower-bound results.

Let us call k to the complexity of the algorithm's used for validity, while $k_\exists$ and $k_\forall$ are the complexity of an **existential** query and the complexity of a **universal** query respectively. $k_{\exists\forall}$ represents both the existential and universal queries have been made.

Then, the complexity of the Brute force algorithm can be reduced to the one explained in Lemma 4.1 below.

Lemma **4.1.** *Let k be the complexity of an existential validity checking procedure, and l the number of literals that the brute force Booleanization algorithm has received. Then, the complexity of this algorithm is as follows:*

$$O(k_\exists \cdot 2^{(k_\exists \cdot 2^p + k_\forall \cdot 2^{nt})}) = O(k_{\exists(\exists \vee \forall)} \cdot 2^{2^l})$$

Note that the method that yields this complexity allows no alternation of quantifiers to occur. This way, we can make use (for both the quantifier elimination over $\bar{y}$ and the validity query over $\bar{x}$) of quantifier elimination procedures that are double exponential on the number of quantifier alternations (see [Grigor'ev and Chistov, 1982]): so both the complexity of $k_\exists$ and $k_\forall$ would be $O(2^{2^{alt}})$, where alt, the number of alternations, is 0. That is: $O(1) \subseteq O(c)$, where c means *a constant*.

4.2.2. Fast algorithms

The second one is a design and implementation of two substantial improvements to our method, based on Cooper's and Ferrante-Rackoff's quantifier elimination methods. The first one (the so-called Minicooper's Booleanization) Booleanizes a fragment of integer literals; whereas the second one, (the so-called Ferrante-Rackoff's Booleanization) Booleanizes rational literals.

Both algorithms are based in the same idea:

1. Forwards pass: Test points (i.e. the partition of the possible choices) of the system are computed.
2. Backwards pass: For each test point, a numeric interval or a witness to represent it is computed.
3. Requirements reconstruction: Using the intervals/witnesses, the partition of the possible choices of the environment is computed (called *decisions*).

We can see an sketch of how this architecture has been implemented for MiniCooper's algorithm in Figure 4.1 below.

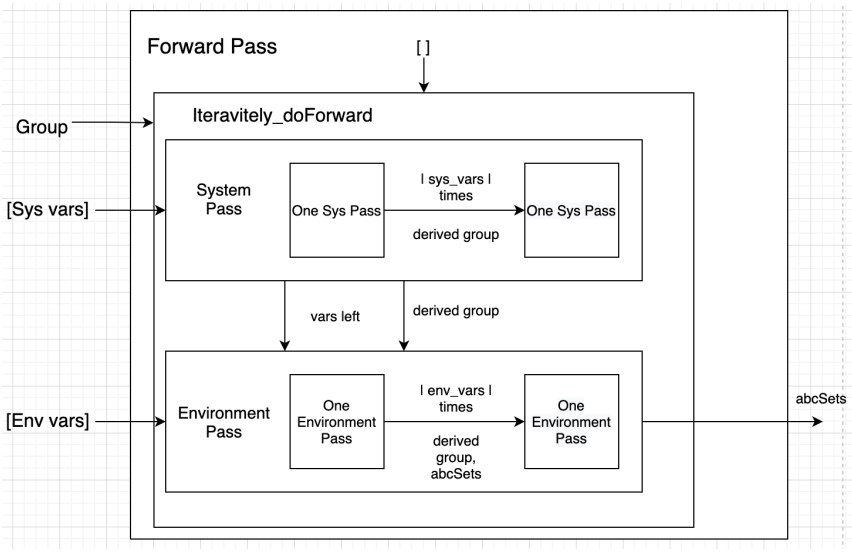

Figure 4.1: The diagram of how the Forwards Pass of Cooper's Booleanization (thus, also MiniCooper) algorithm has been implemented in Ocaml.

As for the complexity analysis of fast algorithms, we conjecture that is lower than the Brute Force one (see Algorithm 1), and also sketch a proof for it. The results are gathered in Claim 4.1 below.

Claim **4.1.** *Let k be the complexity of an existential validity checking procedure, d the number of decisions that a fast Booleanization algorithm has produced and l the number of literals in the original specification. Then, the complexity of fast algorithms is as follows:*

$$O(k \cdot d \cdot 2^l)$$

We also see that, under some conditions, this complexity is considerably decreased. Note also that we have designed some heuristics for both the Brute force and fast algorithms and we conjecture that two of them, a game-based heuristic and a logic-based heuristic, reduce considerably the execution time in practise.

4.3. Upcoming Empirical Analysis

Last but no least, an empirical evaluation of both the brute force algorithm (see Algorithm 1) and the fast ones is about to be performed.

That comparison will be done following some testing parameters when we execute the implementation when given an input. Those parameters are the following ones:

- Testing: We test whether the algorithm has produced an outcome or not (sometimes we except not to). It is marked with '✓' if has so, and '×' if not; and with '?' if it has produced an outcome but is nonsense; for instance, an empty list [].
- Time: We evaluate the resources the algorithm has needed. In this case, we will not focus on the space needed, but on the time, measured in milliseconds (ms).
- Quality: If a ✓result has been given by an algorithm, we measure the quality of it (irrespective to the time). To do so, we have created a metric for quality that measures it numerically.
- Realizability: We use AbsSynthe as the realizability checking tool to see whether the result is **REAL**izable or **UNREAL**izable (and, in case, the result was known manually, compare it with it).

Appendices

A. APPENDIX

Requirements vs Requisites

When talking about *requirements* or *requisites*, there is usually a confusion and misuse.

The words *requisite* and *request* are often used as a synonym for requirement, especially when the speakers are not native English speakers. However, these uses are incorrect in tasks within the IEEE 830-1998 standard.

Requests are used to model customer needs in business language, while requirements are used to model system's features in engineering language. This confusion can come because a *requirement* (out of industrial contexts) is also a synonym of a *need*.

In Table A.1, we can see both confusing definitions (when talking about requirements engineering), and their comparison with English and other languages:

Definition	***All the needs and wishes requested by the customer and the people involved in the development.***	***All the functionalities, features and restrictions that the system should have.***
English	***Request***	***Requirement***
Spanish	*Requerimiento*	*Requisito*
Italian	*Richiedere*	*Requisiti*
French	*Demander*	*Exigence*
German	*Beantragen*	*Anforderung*
Latin	*Requirĕre*	*Requīsitus*

Figure A.1: Comparison between languages on how to say *requirement* and requisite

Therefore, we cannot use the word *requisite* as a synonym of *requirement* and, indeed,

for this reason, it should also be avoided to, for instance, talk about *requisites analysis* as it does not represent what it is are meant to represent: *análisis de requisitos* (Spanish), *Anforderungsanalyse* (German), *analisi dei requisiti* (Italian), *analyse des exigences* (French) and *de requisita analysis* (Latin).

In conclusion, the correct word for this thesis is: *requirement*.

Bibliography

[Arnon et al., 1998] Arnon, D. S., Collins, G. E., and McCallum, S. (1998). Cylindrical algebraic decomposition i: The basic algorithm. In Caviness, B. F. and Johnson, J. R., editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 136–151, Vienna. Springer Vienna.

[Arteche and van der Hallen, 2020] Arteche, N. and van der Hallen, M. (2020). A formal language for qbf family definitions. *KU Leuven*.

[Audemard et al., 2002] Audemard, G., Cimatti, A., Kornilowicz, A., and Sebastiani, R. (2002). Bounded model checking for timed systems. In Peled, D. A. and Vardi, M. Y., editors, *Formal Techniques for Networked and Distributed Sytems — FORTE 2002*, pages 243–259, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Brenguier et al., 2014] Brenguier, R., Pérez, G. A., Raskin, J.-F., and Sankur, O. (2014). Abssynthe: abstract synthesis from succinct safety specifications. *Electronic Proceedings in Theoretical Computer Science*, 157:100–116.

[Camacho et al., 2018] Camacho, A., Bienvenu, M., and McIlraith, S. A. (2018). Finite LTL synthesis with environment assumptions and quality measures. *CoRR*, abs/1808.10831.

[Chaieb, 2006] Chaieb, A. (2006). Verifying mixed real-integer quantifier elimination. *Lecture Notes in Computer Science, 528–540*, page 528–540.

[Clarke and Emerson, 1981] Clarke, E. and Emerson, E. (1981). Design and synthesis of synchronisation skeletons using branching time temporal logic. *Logic of Programs, Proceedings of Workshop*, 132:52–71.

[Cooper, 1972] Cooper, D. W. (1972). Theorem proving in arithmetic without multiplication. *Machine Intelligence*, pages 91–100.

[Ferrante and Rackoff, 1975] Ferrante, J. and Rackoff, X. (1975). A decision procedure for the first order theory of real addition with order. *SIAM Journal of Computation*, page 69–76.

[Grigor'ev and Chistov, 1982] Grigor'ev, D. and Chistov, A. (1982). Polynomial-time factoring of the multivariate polynomials over a global field. *Leningrad, LOMI preprint*.

[Losada, 2020] Losada, N. (2020). Verificación formal de requisitos en sistemas reactivos industriales (translated: Formal verification of requirements in industrial reactive systems). Master's thesis, Universidad del Pais Vasco.

[Markey, 2003] Markey, N. (2003). Temporal logic with past is exponentially more succinct, concurrency column. *Bull. EATCS*, 79:122–128.

[Massey, 1975] Massey, G. J. (1975). Concerning an alleged Sheffer function. *Notre Dame Journal of Formal Logic*, 16(4):549 – 550.

[Monniaux, 2008] Monniaux, D. (2008). A quantifier elimination algorithm for linear real arithmetic.

[Nipkow, 2008] Nipkow, T. (2008). Linear quantifier elimination. *Automated Reasoning. IJCAR 2008. Lecture Notes in Computer Science, vol 5195*.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, page 46–57.

[Pnueli and Rosner, 1989] Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, page 85–101.

[Post, 1941] Post, E. L. (1941). The two-valued iterative systems of mathematical logic. *Annals of Mathematics studies, no. 5, 122pp*.

[Presburger, 1929] Presburger, M. (1929). Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen,in welchem die addition als einzige operation hervortritt (translated: On the completeness of a certain system of arithmetic of integers, in which addition stands out as the only operation). *Warsaw: in Comptes Rendus du I congrès de Mathématiciens des Pays Slaves.*, pages 92–101.

[Sturm, 2017] Sturm, T. (2017). A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Mathematics in Computer Science*, 11.

[Suriana, 2016] Suriana, P. (2016). *Fourier-Motzkin with non-linear symbolic constant coefficients*. PhD thesis, Massachusetts Institute of Technology.

[Tonetta, 2017] Tonetta, S. (2017). Linear-time temporal logic with event freezing functions. *Electronic Proceedings in Theoretical Computer Science*, 256.

[Weispfenning, 1997] Weispfenning, V. (1997). Quantifier elimination for real algebra—the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, page 85–101.

[Wesselkamper, 1975] Wesselkamper, T. C. (1975). A sole sufficient operator. *Notre Dame Journal of Formal Logic*, 16(1):86 – 88.

[Zimmermann and Klein, 2014] Zimmermann, M. and Klein, F. (2014). Infinite games, lecture notes. *Saarland university*.