TÉCNICO LISBOA

DTL: Translation, SMT Verification, Separation and Interpolation

Miguel de Lacerda e Costa Serra do Nascimento

Thesis to obtain the Master of Science Degree in

Mathematics and Applications

Supervisor(s): Prof. Jaime Arsénio de Brito Ramos
Prof. João Filipe Quintas dos Santos Rasga

January 2021

Declaração

Declaro que o presente documento é um trabalho original da minha autoria e que cumpre todos os requisitos do Código de Conduta e Boas Práticas da Universidade de Lisboa.

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank Professor Jaime Ramos and Professor João Rasga for accepting to supervise this thesis and for all the helpful guidance and insight given throughout its development.

I am grateful to my parents for all the support they have given me throughout my education and for everything they have done for me.

Resumo

O objetivo desta dissertação é contribuir para uma melhor compreensão da Lógica Temporal Distribuída (DTL), nomeadamente ao investigar se possui algumas propriedades lógicas importantes.

Começamos por apresentar uma tradução de fórmulas em DTL para fórmulas em lógica de primeira ordem (FOL) que preserva a consequência semântica, e de seguida recorremos a CVC4, um provador de teoremas e solucionador de satisfiability modulo theories (SMT), de forma a verificar a validade de fórmulas em DTL, capitalizando na tradução de DTL para FOL mencionada anteriormente. Também propomos uma extensão da propriedade da separação a DTL, e provamos que a lógica temporal distribuída cujas linguagens locais contêm os operadores Until e Since tem a nossa versão da propriedade da separação. Adaptamos ainda a propriedade da interpolação de Craig a DTL, e provamos que um fragmento desta lógica tem a propriedade referida.

Palavras-chave: Lógica temporal distribuída, CVC4, propriedade da separação, interpolação de Craig

Abstract

The purpose of this dissertation is to contribute to better understand Distributed Temporal Logic (DTL), namely by investigating whether it enjoys some important logical properties.

We start by presenting a translation from DTL formulas into first-order logic (FOL) formulas that preserves entailment, and afterwards we resort to the theorem prover and satisfiability modulo theories (SMT) solver CVC4 in order to check the validity of DTL formulas, capitalizing on the translation from DTL to FOL previously mentioned. Furthermore, we propose an extension of the separation property to DTL, and we prove that our extension of this property holds for the distributed temporal logic whose local languages contain both the Until and Since operators. We also adapt the Craig interpolation property to DTL, and we prove that the property holds for a fragment of this logic.

Keywords: Distributed temporal logic, CVC4, separation property, Craig interpolation

Contents

List of Figures

Chapter 1

Introduction

Distributed temporal logic (DTL) is a temporal logic introduced with the purpose of reasoning about temporal properties of discrete distributed systems from the local point of view of its agents [1, 2]. Having been first proposed near the end of the twentieth century in [3], DTL is a relatively recent logic, when comparing to, for instance, first-order logic (FOL) or linear temporal logic (LTL). Because of this, there are still some important ideas and logical properties left to be studied for DTL, some of which may improve the applicability of this logic in certain areas. The purpose of this document is to investigate whether DTL enjoys some of these logical properties.

In this work, we will start by presenting the syntax and semantics of FOL, LTL and DTL, and we will then proceed to approach the topic of translation. More specifically, we will present a translation from DTL formulas into FOL formulas that preserves entailment. In fact, some problems can be made easier to solve by considering the DTL formula translated into FOL, rather than the DTL formula itself. We will make use of this translation when studying satisfiability modulo theories (SMT) verification, since the question of whether a DTL formula is valid or not can be tied to the SMT problem.

The SMT problem is a variant of the SAT problem for which the non-logical symbols are interpreted in the context of some background theory. We will study CVC4, a theorem prover for SMT problems, and present its techniques for dealing with quantified SMT instances. We will finish the topic of verification when attempting to check, using CVC4, the validity of DTL formulas, by initially translating them into FOL.

The topic of separation will be addressed as well. A logic is said to have the property of separation if every formula is equivalent to a Boolean combination of formulas that each refer only to the present, past or future [4–7]. Dov Gabbay was the first to show that the temporal logic with the operators Until and Since has the separation property over the integers [8]. It turns out that, for temporal logic, the notion of separation is tied to the expressiveness of the logic, that is, the variety and quantity of ideas that the logic can be used to represent [9, 10]. In this work, a proposal for an extension of the separation property to distributed temporal logic is presented, along with the proof that this extension of the separation property holds for the distributed temporal logic whose local languages contain both the Until and Since operators.

We will finish this document with the topic of Craig interpolation. A logic having the Craig interpolation property is such that if a formula ϕ entails a formula ψ, then there exists a formula θ (called the interpolant) such that ϕ entails θ, θ entails ψ, and every propositional symbol in θ occurs both in ϕ and ψ. When it comes to applications in computer science, interpolation is often a desired property to have in a temporal logic. For example, interpolation has played a role in building efficient model checkers. Uniform interpolation, which we will talk about in this work, has been particularly useful in this regard.

The Craig interpolation property is proven to hold for both FOL and a fragment of LTL [11, 12]. We will study the Craig interpolation property in the context of distributed temporal logic, and prove that this property holds for the fragment of DTL whose local languages contain X as the only temporal operator.

1.1 Thesis Outline

To sum up, this document is organized as follows:

- In Chapter 2, we introduce the syntax and semantics of FOL, LTL and DTL. We also present a translation from DTL formulas to FOL formulas, and we prove that it preserves entailment.
- In Chapter 3, we study the SMT solver CVC4 and the techniques it utilizes for dealing with quantified formulas. We also attempt to check the validity of LTL and DTL formulas using CVC4, capitalizing on the previous translation from DTL to FOL.
- In Chapter 4, we propose an extension of the separation property to distributed temporal logic, and we prove that this extension of the property holds for the distributed temporal logic whose local languages contain both the Until and Since operators.
- In Chapter 5, we prove that the Craig interpolation property holds for the distributed temporal logic whose local languages contain X as the only temporal operator.
- In Appendix A, we transcript the code developed in Mathematica for translating LTL and DTL formulas into FOL formulas that are written in CVC4's native input language.

Chapter 2

A Translation from DTL into FOL

In this chapter, we present the syntax and semantics of first-order logic (FOL), linear temporal logic (LTL) and distributed temporal logic (DTL). Furthermore, we aim to show how to translate from DTL into FOL. With this intention, we will define a translation function that translates DTL formulas into FOL formulas, and prove that our translation function preserves entailment in DTL. This translation function will be necessary for section 3.3, where we attempt to check the validity of DTL formulas by resorting to a Mathematica function that, based on the translation function we will define, translates DTL formulas into FOL formulas written in CVC4's native language. The code of this Mathematica function, along with some guidance on how to use this function, can be found in Appendix A.

Finally, the translation function will also be expanded to other temporal operators than are not considered in the definition of the function.

2.1 First-order Logic

2.1.1 Syntax

We will introduce formulas in the context of FOL. First, we start by explaining what a first-order signature is.

Definition 2.1. A *first-order signature* is a tuple $\Sigma = \langle \mathcal{F}, \mathcal{P}, \tau \rangle$ such that

- $\mathcal{F}$ and $\mathcal{P}$ are disjoint sets, with $\mathcal{P} \neq \emptyset$;
- $\tau : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ is a map.

The elements of $\mathcal{F}$ are said to be the *function symbols*, while the elements of $\mathcal{P}$ are said to be the *predicate symbols* (or predicate letters). The map τ returns the arity of its argument. Additionally, let

- $\mathcal{F}_n$ denote the subset of function symbols with arity n;
- $\mathcal{P}_n$ denote the subset of predicate symbols with arity n.

Example 2.1. Let us consider a first-order signature Σ, where $\mathcal{F} = \{0, 1, +, \times\}$ and $\mathcal{P} = \{\cong\}$. Then, we have that $\mathcal{F}_0 = \{0, 1\}$, $\mathcal{F}_2 = \{+, \times\}$ and $\mathcal{P}_2 = \{\cong\}$.

Before introducing formulas in first-order logic, we first have to say what is a term. Let $\mathcal{X} = \{x_0, x_1, \dots\}$ denote the set of variables.

Definition 2.2. The set $\mathcal{T}_\Sigma$ of *terms* over Σ is inductively defined as follows:

- $\mathcal{F}_0 \cup \mathcal{X} \subseteq \mathcal{T}_\Sigma$;
- $f(t_1, \dots, t_n) \in \mathcal{T}_\Sigma$ provided that $f \in \mathcal{F}_n$ and $t_1, \dots, t_n \in \mathcal{T}_\Sigma$.

Definition 2.3. The set $\mathcal{L}_\Sigma$ of *formulas* over Σ is inductively defined as follows:

- $\perp \in \mathcal{L}_\Sigma$ - "bottom", denotes a proposition that is always false;
- $p(t_1, \dots, t_n) \in \mathcal{L}_\Sigma$ provided that $p \in \mathcal{P}_n$ and $t_1, \dots t_n \in \mathcal{T}_\Sigma$;
- $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{L}_\Sigma$ provided that $\varphi_1, \varphi_2 \in \mathcal{L}_\Sigma$ - implication;
- $\forall x \, \varphi \in \mathcal{L}_\Sigma$ provided that $x \in \mathcal{X}$ and $\varphi \in \mathcal{L}_\Sigma$ - universal quantification.

For the sake of simplicity, it may be useful to consider certain formula abbreviations. As such, we define the following abbreviations:

- $\neg \, \varphi \equiv (\varphi \Rightarrow \perp)$ - negation;
- $\top \equiv (\perp \Rightarrow \varphi)$ - denotes a proposition which is unconditionally true;
- $\varphi \vee \psi \equiv \neg \, \varphi \Rightarrow \psi$ - disjunction;
- $\varphi \wedge \psi \equiv \neg \, (\varphi \Rightarrow \neg \psi)$ - conjunction;
- $\exists x \, \varphi \equiv \neg \, \forall x \, \neg \, \varphi$ - existential quantification.

In a formula, it may also be important to distinguish between variables that occur free and variables that are bounded to a quantifier.

We inductively define the map var_Σ that assigns to each term the set of variables occurring in it the following way:

- $var_\Sigma(x) = \{x\}$;
- $var_\Sigma(c) = \emptyset$;
- $var_\Sigma(f(t_1, \dots, t_n)) = var_\Sigma(t_1) \cup \dots \cup var_\Sigma(t_n)$.

Also, we inductively define the map fv_Σ that assigns to each formula the set of *variables occurring free* in it in the following way:

- $fv_\Sigma(p(t_1, \dots, t_n)) = var_\Sigma(t_1) \cup \dots \cup var_\Sigma(t_n)$.

- $fv_\Sigma(\perp) = \emptyset$;
- $fv_\Sigma(\varphi \Rightarrow \psi) = fv_\Sigma(\varphi) \cup fv_\Sigma(\psi)$;
- $fv_\Sigma(\forall x\, \varphi) = fv_\Sigma(\varphi) \backslash \{x\}$.

Note the following example:

$$fv_\Sigma((\exists x_1\, p(x_1, x_2)) \wedge (\forall x_3\, q(x_1, x_3))) = \{x_1, x_2\}.$$

2.1.2 Semantics

In this subsection, we study the semantics of FOL, that is, the study of the logical system in the point of view of their interpretation. With this in mind, we introduce the concepts of interpretation structure, assignment and satisfaction.

Definition 2.4. An *interpretation structure* over Σ is a tuple $\mathcal{I} = \langle \mathcal{D}, \{f^{\mathcal{I}}\}_{f \in \mathcal{F}}, \{p^{\mathcal{I}}\}_{p \in \mathcal{P}} \rangle$ such that

- $\mathcal{D}$ is a non-empty set which we call the domain;
- $f^{\mathcal{I}} : \mathcal{D}^n \to \mathcal{D}$ is a map providing that $f \in \mathcal{F}_n$;
- $p^{\mathcal{I}} : \mathcal{D}^n \to \{0, 1\}$ is a map providing that $p \in \mathcal{P}_n$;

Essentially, $f^{\mathcal{I}}$ and $p^{\mathcal{I}}$ represent the interpretation or denotation of, respectively, f and p in $\mathcal{I}$.

Example 2.2. Let Σ be a first-order signature, where $\mathcal{F} = \{0, 1, +, \times\}$ and $\mathcal{P} = \{\cong\}$. We can consider an interpretation structure $\mathcal{I}$ over Σ such that $\mathcal{I} = \langle \mathbb{Q}, \{0^{\mathcal{I}}, 1^{\mathcal{I}}, +^{\mathcal{I}}, \times^{\mathcal{I}}\}, \{\cong^{\mathcal{I}}\} \rangle$ and

- $0^{\mathcal{I}} = 0$;
- $1^{\mathcal{I}} = 1$;
- $+^{\mathcal{I}} : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, such that $d_1 +^{\mathcal{I}} d_2 = d_1 + d_2$;
- $\cong^{\mathcal{I}} (d_1, d_2) = 1$ provided that $d_1 = d_2$.

An assignment is a map $\rho : \mathcal{X} \to \mathcal{D}$. We say that an assignment σ is x-equivalent to another assignnment ρ, which we write $\sigma \equiv_x \rho$, if $\sigma(y) = \rho(y)$ for every $y \in \mathcal{X} \backslash \{x\}$.

Definition 2.5. We inductively define *contextual satisfaction* as follows:

- $\mathcal{I}\rho \nVdash_\Sigma \perp$;
- $\mathcal{I}\rho \Vdash_\Sigma p(t_1, \ldots, t_n)$ whenever $p^{\mathcal{I}}([\![t_1^{\mathcal{I}\rho}]\!], \ldots, [\![t_n^{\mathcal{I}\rho}]\!]) = 1$;
- $\mathcal{I}\rho \Vdash_\Sigma \varphi_1 \Rightarrow \varphi_2$ provided that either $\mathcal{I}\rho \nVdash_\Sigma \varphi_1$ or $\mathcal{I}\rho \Vdash_\Sigma \varphi_2$;
- $\mathcal{I}\rho \Vdash_\Sigma \forall x\, \varphi$ providing that $\mathcal{I}\sigma \Vdash_\Sigma \varphi$ for every σ that is x-equivalent to ρ.

Given an interpretation structure $\mathcal{I}$ and an assignment ρ, if $\mathcal{I}\rho \Vdash \varphi$, we say that $\mathcal{I}$ and ρ contextually satisfy $\varphi \in \mathcal{L}_\Sigma$.

Definition 2.6. We say that $\mathcal{I}$ *satisfies* φ (we can also say that φ is true in $\mathcal{I}$, or that $\mathcal{I}$ is a model of φ), which we write $\mathcal{I} \Vdash_\Sigma \varphi$, whenever $\mathcal{I}\rho \Vdash_\Sigma \varphi$ for every ρ.

Definition 2.7. A formula is *valid*, written $\vDash_\Sigma \varphi$, if $\mathcal{I} \Vdash \varphi$ for every interpretation structure $\mathcal{I}$ over Σ.

We will also need the definition of entailment and the definition of theory for some of the following sections. We introduce these concepts in the next definitions.

Definition 2.8. A formula φ over Σ is *entailed* by a set Γ of formulas over the same signature, which we write $\Gamma \vDash_\Sigma \varphi$, if, for every interpretation structure $\mathcal{I}$ over Σ, $\mathcal{I} \Vdash_\Sigma \varphi$ whenever $\mathcal{I} \Vdash_\Sigma \gamma$ for each $\gamma \in \Gamma$.

Definition 2.9. The *semantic closure* of a set $\Gamma \subseteq \mathcal{L}_\Sigma$ is the set

$$\Gamma^{\vDash_\Sigma} = \{\varphi \in \mathcal{L}_\Sigma : \Gamma \vDash_\Sigma \varphi\}$$

of its entailed formulas.

Definition 2.10. A set of formulas $\Theta \subseteq L_\Sigma$ is said to be a *theory* if $\Theta^{\vDash_\Sigma} = \Theta$.

2.2 Linear Temporal Logic

While first-order logic describes a static situation, temporal logic is able to depict that situation as time progresses. In particular, linear temporal logic (LTL) is a propositional temporal logic with modalities that describe events along a single time path.

2.2.1 Syntax

We introduce LTL formulas in the following definition.

Definition 2.11. Given a set of propositional symbols $Prop$, we define the language of linear temporal logic $\mathcal{L}_{LTL}$ as follows:

$$\mathcal{L}_{LTL} ::= Prop \mid \bot \mid \mathcal{L}_{LTL} \Rightarrow \mathcal{L}_{LTL} \mid \mathsf{X}\,[\mathcal{L}_{LTL}] \mid \mathcal{L}_{LTL}\,\mathsf{U}\,\mathcal{L}_{LTL} \mid \mathcal{L}_{LTL}\,\mathsf{S}\,\mathcal{L}_{LTL}.$$

The $\bot$ and $\Rightarrow$ have the usual meanings. Intuitively, the formula $\mathsf{X}\varphi$, which we call a next formula, stands for "φ will hold in the next instant". The formula $\varphi_1\,\mathsf{U}\,\varphi_2$, which we call a until formula, stands for "there is an instant in the future where φ_2 will hold and until then formula φ_1 must hold". Furthermore, the formula $\varphi_1\,\mathsf{S}\,\varphi_2$, which we call a since formula, is similar to an until formula but in the past direction. Intuitively, it stands for "since the last instant in the past for which φ_2 was true, φ_1 has always been true".

We introduce the following abbreviations:

- $\mathsf{F}\,\varphi \equiv \top\,\mathsf{U}\,\varphi$ - which stands for "sometime in the future, φ must hold";

- $\mathsf{G}\,\varphi \equiv \neg\,\mathsf{F}\,\neg\,\varphi$ - which stands for “always in the future, φ must hold”.

Note that all the temporal operators we have shown talk about the future, except for the since operator, which talks about the past. Additionally, temporal operators can be combined to express more complex properties. We will see some examples of both of these situations in the next sections and chapters.

2.2.2 Semantics

First, we introduce the definition of finite and infinite words. We assume a nonempty and finite set Σ, called the alphabet, whose elements are called symbols or letters.

Definition 2.12. Let Σ be an alphabet. A *finite word* w over Σ is a finite, possibly empty, sequence $\nu_1\nu_2\ldots\nu_n$ where $n \in \mathbb{N}$ and each $\nu_i \in \Sigma$, for $i = 1, \ldots, n$. The set of all finite words over Σ is denoted by Σ^*.

The finite word corresponding to the empty sequence, which we call the empty word, is denoted by ϵ. The length of a word, which we denote by $|w|$, is the number of symbols that appear in the sequence. For instance, the length of the word $w = \nu_1\nu_2\ldots\nu_n$ in n. The length of the empty word is 0. Also, we write $w|_i$ to denote the prefix of w of length i, that is, $w|_i = \nu_1\ldots\nu_i$, provided that $0 \leq i \leq |w|$.

Definition 2.13. An *infinite word* σ over Σ is an infinite sequence $\sigma = \nu_1\nu_2\ldots$ where each $\nu_i \in \Sigma$, for $i \in \mathbb{N}$. The set of all infinite words over Σ is denoted by Σ^ω.

The length of an infinite word is always ω.

Definition 2.14. An *interpretation for LTL* is an infinite word over 2^{Prop}.

Intuitively, an interpretation structure for LTL is an infinite sequence of valuations, such that each element of the sequence in an LTL interpretation structure will determine the boolean values of the propositional symbols at a given moment in time. Given an interpretation $\sigma = \nu_0\nu_1\nu_2\ldots$, we can see ν_i as the set of propositional symbols that hold at an instant i. We can now define satisfaction in the context of LTL.

Definition 2.15. Let σ be an interpretation and $i \in \mathbb{N}$. The local satisfaction relation for LTL is inductively defined as follows:

- $\sigma, i \nVdash \bot$;
- $\sigma, i \Vdash p$ if $p \in \sigma[i]$ if $\sigma_i(p) = 1$;
- $\sigma, i \Vdash \varphi_1 \Rightarrow \varphi_2$ if $\sigma, i \nVdash \varphi_1$ or $\sigma, i \Vdash \varphi_2$;
- $\sigma, i \Vdash \mathsf{X}\,\varphi$ if $\sigma, i+1 \Vdash \varphi$;
- $\sigma, i \Vdash \varphi_1 \,\mathsf{U}\, \varphi_2$ if there is $j > i$ such that $\sigma, j \Vdash \varphi_2$ and $\sigma, k \Vdash \varphi_1$, for every $i < k < j$.

The interpretation σ satisfies the formula φ, which we write $\sigma \Vdash \varphi$, if $\sigma, i \Vdash \varphi$ for every i. Similarly to FOL, we say that a formula is valid if it is satisfied by all interpretations.

We also define entailment in the context of LTL.

Definition 2.16. Let $\Gamma \cup \{\varphi\} \subseteq \mathcal{L}$. We say that Γ entails φ, written $\Gamma \vDash \varphi$, when $\sigma \Vdash \varphi$ for every interpretation σ such that $\sigma \Vdash \Gamma$.

2.3 Distributed Temporal Logic

2.3.1 Syntax

The syntax of distributed temporal logic (DTL) is defined over a distributed signature.

Definition 2.17. A *distributed signature* is defined as a tuple $\Sigma = \langle Id, \{Prop_i\}_{i \in Id} \rangle$, where Id is a non-empty finite set of agents and, for each $i \in Id$, $Prop_i$ is a set of local state propositions.

The global language $\mathcal{L}_{DTL}$ is defined by

$$\mathcal{L}_{DTL} ::= @_{i_1}[\mathcal{L}_{i_1}] \mid \cdots \mid @_{i_n}[\mathcal{L}_{i_n}] \mid \bot \mid \mathcal{L}_{DTL} \Rightarrow \mathcal{L}_{DTL},$$

for $Id = \{i_1, \ldots, i_n\}$, where the local languages $\mathcal{L}_i$ for each $i \in Id$ are defined by

$$\mathcal{L}_i ::= Prop_i \mid \bot \mid \mathcal{L}_i \Rightarrow \mathcal{L}_i \mid \mathcal{L}_i \, \mathsf{U} \, \mathcal{L}_i \mid \mathcal{L}_i \, \mathsf{S} \, \mathcal{L}_i \mid ©_j[\mathcal{L}_j],$$

with $j \in Id$.

The $\bot$ and $\Rightarrow$ have the usual meaning, while U and S were introduced in the previous section. In the global language, we introduce a new kind of formula. We say that the formula $@_i[\varphi]$, called a global formula, means that φ holds for agent i. On the other hand, local formulas hold locally for each agent.

Note that temporal operators only occur in local formulas, meaning that we talk about these formulas in the context of a given agent. Also locally for an agent i, the formula $©_j[\psi]$, called a communication formula, means that agent i has just communicated (or synchronized) with agent j, for whom ψ holds.

2.3.2 Semantics

Before presenting the definition of a DTL interpretation structure, we will introduce have to introduce the concepts of local and distributed life-cycles, and of local and global states.

Definition 2.18. A *local life-cycle* of an agent $i \in Id$ is a countable infinite, discrete and well-founded total order $\lambda_i = \langle Ev_i, \leq_i \rangle$, where Ev_i is the set of local events and $\leq_i$ the local order of causality.

Definition 2.19. The relation $\rightarrow_i \subseteq Ev_i \times Ev_i$, called the *local successor relation*, is the relation such that $e \rightarrow_i e'$ if $e <_i e'$ and there is no e'' such that $e <_i e'' <_i e'$.

As a consequence, $\leq_i = \rightarrow_i^*$, i.e., $\leq_i$ is the reflexive, transitive closure of $\rightarrow_i$.

Definition 2.20. A *distributed life-cycle* is a family $\lambda = \{\lambda_i\}_{i\in Id}$ of local life-cycles. This family is such that $\leq = (\bigcup_{i\in Id} \leq_i)^*$ defines a partial order of global causality on the set of all events $E = \bigcup_{i\in Id} E_i$.

Note that, due to the fact that communication between agents involves event sharing, we may have, for some event e, $e \in E_i \cap E_j$, for $i \neq j$.

Definition 2.21. The *local state* of agent i is a finite set $\xi_i \subseteq Ev_i$ down-closed for local causality, that is, if $e \leq_i e'$ and $e' \in \xi_i$, then also $e \in \xi_i$.

We denote the set of all local states of an agent i by Ξ_i. This set is totally ordered by inclusion and has $\emptyset$ as the minimal element.

Due to the total order on local events, the local states of each agent are totally ordered. The 0^{th} state of each agent is $\emptyset$, and the next local state is reached by the occurrence of an event which we call $last(\xi_i)$, since it is the last event in which agent i took part in order to reach the present state ξ_i. We denote by ξ_i^k the k^{th} state of agent i, meaning that $\xi_i^0 = \emptyset$ is the initial state and ξ_i^k is the state reached after the occurrence of the first k events. Note that ξ_i^k is the only state of agent i that contains exactly k elements, that is, where $|\xi_i^k| = k$. Moreover, given $e \in Ev_i$, $(e \downarrow i) = \{e' \in Ev_i | e' \leq_i e\}$ is always a local state. Furthermore, we have that $(last(\xi_i) \downarrow i) = \xi_i$, assuming that ξ_i is non-empty.

Definition 2.22. A *global state* is a finite set $\xi \subseteq Ev$ closed for global causality, that is, if $e \leq e'$ and $e' \in \xi$, then also $e \in \xi$.

The set of all global states, written Ξ, has $\emptyset$ as the minimal element. Also, we can see that every global state ξ includes the local state of agent i.

After these definitions, we can finally introduce DTL interpretation structures.

Definition 2.23. An *interpretation structure* μ *for DTL* is a labelled distributed life-cycle of the form $\mu = \langle \lambda, \sigma \rangle$, where λ consists of a distributed life-cycle and $\sigma = \{\sigma_i\}_{i\in Id}$ is a family of local labelling functions.

For each $i \in Id$, the local labelling functions σ_i associate a set of local state propositions to each local state. Furthermore, we also denote the tuple $\langle \lambda_i, \sigma_i \rangle$ by μ_i.

Now, we can define the global satisfaction relation by

- $\mu \Vdash \gamma$ if $\mu, \xi \Vdash \gamma$ for every $\xi \in \Xi$,

where the global satisfaction relation at a global state is defined by

- $\mu, \xi \nVdash \perp$;
- $\mu, \xi \Vdash \gamma \Rightarrow \delta$ if $\mu, \xi \nVdash \gamma$ or $\mu, \xi \Vdash \delta$;
- $\mu, \xi \Vdash @_i[\varphi]$ if $\mu_i \Vdash_i \varphi$ if $\mu_i, \xi \Vdash_i \varphi$ for every $\xi \in \Xi_i$,

and where the local satisfaction relations at local states are defined by

- $\mu_i, \xi \Vdash_i p$ if $p \in \sigma_i(\xi)$;

- $\mu_i, \xi \Vdash_i \neg\varphi$ if $\mu_i, \xi \nVdash_i \varphi$;

- $\mu_i, \xi \Vdash_i \varphi \Rightarrow \psi$ if $\mu_i, \xi \nVdash_i \varphi$ or $\mu_i, \xi \Vdash_i \psi$;

- $\mu_i, \xi \Vdash_i \varphi \,\mathsf{U}\, \psi$ if $|\xi| = k$ and there exists $\xi_i^n \in \Xi_i$ such that $k < n$ with $\mu_i, \xi_i^n \Vdash_i \psi$, and $\mu_i, \xi_i^m \Vdash_i \varphi$ for every $k < m < n$;

- $\mu_i, \xi \Vdash_i \varphi \,\mathsf{S}\, \psi$ if $|\xi| = k$ and there exists $\xi_i^n \in \Xi_i$ such that $n < k$ with $\mu_i, \xi_i^n \Vdash_i \psi$, and $\mu_i, \xi_i^m \Vdash_i \varphi$ for every $n < m < k$;

- $\mu_i, \xi \Vdash_i ©_j[\varphi]$ if $|\xi| > 0, last_i(\xi) \in E_j$, and $\mu_j, (last_i(\xi) \downarrow j) \Vdash_j \varphi$.

A DTL formula γ is said to be valid, written $\vDash \gamma$, if $\mu \Vdash \gamma$ for every global interpretation structure μ.

Finally, we define entailment in the context of DTL.

Definition 2.24. Let $\Gamma \cup \{\varphi\} \subseteq \mathcal{L}_{DTL}$. We say that Γ entails φ, written $\Gamma \vDash \varphi$, when $\sigma \Vdash \varphi$ for every interpretation σ such that $\sigma \Vdash \Gamma$.

2.4 The Translation Function

In this section, we present a translation function from DTL formulas into FOL formulas, and we show that this translation preserves entailment in DTL.

First, we need to introduce some definitions. To start, note that, excluding communication formulas, local DTL formulas coincide with LTL formulas. This fact is used in [2] to prove that DTL is decidable by a translation into LTL. We make use of this idea, along with the fact that there is a known translation from LTL into FOL [13], in our own translation.

Given a DTL signature $\Sigma = \langle Id, Prop\rangle$, we define the corresponding FOL signature as having $\mathcal{P} = \{@(i,n) \mid i \in Id,\, n \in \mathbb{N}_0\} \cup \biguplus_{i\in Id} Prop_i \cup \{<\}$, where $<$ has the usual meaning. We assume that the symbol $p \in Prop_i$ is represented in $\mathcal{P}$ by the unary predicate p_i. The additional predicate $@(i,n)$, with $i \in Id$ and $n \in \mathbb{N}_0$, is meant to express whether the n-th event in the global order of events of the DTL signature belongs to agent i or not.

Thus, the translation of global formulas is given by the function $f : \mathcal{L}_{DTL} \to \mathcal{L}_{FOL}$ such that

- $f(@_i[\varphi]) = \forall x\ (@(i,x) \Rightarrow f_i(\varphi, x))$,

and for each $i \in Id$, the function $f_i : \mathcal{L}_i \to \mathcal{L}_{FOL}$ translates local formulas to FOL formulas the following way:

- $f_i(p, x) = p_i(x)$;

- $f_i(\neg\varphi, x) = \neg f_i(\varphi, x)$;

- $f_i(\varphi \Rightarrow \psi, x) = f_i(\varphi, x) \Rightarrow f_i(\psi, x)$;

- $f_i(\varphi \,\mathsf{U}\, \psi, x) = \exists y\ x < y \wedge @(i,y) \wedge f_i(\psi, y) \wedge \forall z\ (x < z < y \Rightarrow (@(i,z) \Rightarrow f_i(\varphi, z)))$;

- $f_i(\varphi \mathsf{S} \psi, x) = \exists y\, y < x \wedge @(i,y) \wedge f_i(\psi, y) \wedge \forall z\, (y < z < x \Rightarrow (@(i,z) \Rightarrow f_i(\varphi, z)))$;
- $f_i(©_j[\varphi], x) = @(j,x) \wedge f_j(\varphi, x)$.

Let us also consider the map β from FOL interpretation structures to DTL interpretation structures such that $\beta(\mathcal{I}) = \langle \lambda, \sigma \rangle$, with $\lambda_i = \langle E_i, \leq_i \rangle$, where:

- $E_i = \{n \in \mathbb{N} \mid @(i,n) \in \mathcal{P}^{\mathcal{I}}\}$;
- $\leq_i$ is the restriction of the usual order on $\mathbb{N}$, with $n \to_i m$ if $n, m \in E_i$ and there is no $k \in E_i$, such that $n < k < m$;
- $\sigma_i(\emptyset) = \{p \in Prop_i \mid p_i(0) = 1\}$ and $\sigma_i(\{m \in E_i \mid m \leq n\}) = \{p \in Prop_i \mid p_i(n) = 1\}$, for each $n \in E_i$.

Now, we introduce and prove Proposition 2.1, which is necessary for showing that our functions f_i, for $i \in Id$, are well-defined. Note that we must only consider the FOL interpretation structures that satisfy $\{\bigwedge_{i \in Id} @(i,0)\}$. We need to add this restriction due to the fact that, at the initial DTL state $\emptyset$, no events have yet occurred.

Proposition 2.1. Given a FOL interpretation structure $\mathcal{I}$ that satisfies $\{\bigwedge_{i \in Id} @(i,0)\}$, we have that, for every $\varphi \in \mathcal{L}_{DTL}$,

$$\beta(\mathcal{I})_i, \xi_i^k \Vdash_i \varphi \text{ if and only if } \mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(\varphi, x), \text{ for every } \xi_i^k \in \Xi_i,$$

where $[x/last_i(\xi_i^k)]$ stands for a variable assignment that assigns the free variable x of $f_i(\varphi, x)$ the value $last_i(\xi_i^k)$.

Proof. The proof follows by induction on φ. We assume that $last_i(\emptyset) = 0$. For the basis of induction, if the formula is a propositional symbol p, then $\beta(\mathcal{I})_i, \xi_i^k \Vdash_i p$ iff $p \in \sigma_i(\xi_i^k)$ iff $p_i(last_i(\xi_i^k)) = 1$ iff $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(p, x)$. As for the induction step, let us consider the following cases:

- The formula is $\neg\varphi$. Then, $\beta(\mathcal{I})_i, \xi_i^k \Vdash \neg\varphi$ iff $\beta(\mathcal{I})_i, \xi_i^k \nVdash \varphi$ iff, by the induction hypothesis, $\mathcal{I}, [x/last_i(\xi_i^k)] \nVdash_{FOL} f_i(\varphi, x)$ iff $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} \neg f_i(\varphi, x)$ iff $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(\neg\varphi, x)$.
- The formula is $\varphi \Rightarrow \psi$. If we assume that $\mathcal{I}, [x/last_i(\xi_i^k)] \nVdash_{FOL} f_i(\varphi \Rightarrow \psi, x)$, this leads to $\mathcal{I}, [x/last_i(\xi_i^k)] \nVdash_{FOL} f_i(\varphi, x) \Rightarrow f_i(\psi, x)$, which in turn implies that $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(\varphi, x)$ and $\mathcal{I}, [x/last_i(\xi_i^k)] \nVdash_{FOL} f_i(\psi, x)$. By the induction hypothesis, we have that $\beta(\mathcal{I})_i, \xi_i^k \Vdash_i \varphi$ and $\beta(\mathcal{I})_i, \xi_i^k \nVdash_i \psi$. Therefore, $\beta(\mathcal{I})_i, \xi_i^k \nVdash_i \varphi \Rightarrow \psi$. The converse is similar.
- The formula is $\varphi \mathsf{U} \psi$. If we assume that $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(\varphi \mathsf{U} \psi, x)$, then there exists $y > last_i(\xi_i^k)$ such that $\mathcal{I}, [x/y] \Vdash_{FOL} @(i,x) \wedge f_i(\psi, x)$. Therefore, $y \in E_i$ and $(y \downarrow i) = \xi_i^n$ for some $n > k$, which means that $last_i(\xi_i^n) = y$. Hence $I, [x/last_i(\xi_i^n)] \Vdash_{FOL} f_i(\psi, x)$, and by the induction hypothesis, we get that $\beta(\mathcal{I})_i, \xi_i^n \Vdash_i \psi$. Furthermore, $\mathcal{I}, [x/z] \Vdash_{FOL} @(i,x) \Rightarrow f_i(\varphi, x)$ for every z such that $last_i(\xi_i^k) < z < y$. Given m such that $k < m < n$, we have that $last_i(\xi_i^k) < last_i(\xi_i^m) < last_i(\xi_i^n) = y$. Since $last_i(\xi_i^m) \in E_i$, it follows that $\mathcal{I}, [x/last_i(\xi_i^m) \Vdash_{FOL} @(i,x)$ and, therefore,

$\mathcal{I}, [x/last_i(\xi_i^m)] \Vdash_{FOL} f_i(\varphi, x)$. By the induction hypothesis, we obtain $\beta(\mathcal{I})_i, \xi_i^m \Vdash_i \varphi$. Thus, we conclude that $\beta(\mathcal{I})_i, \xi_i^m \Vdash_i \varphi \mathsf{U} \psi$. The converse is similar.

- The proof for S is similar to the proof for U.

- Assume that the formula is $©_j[\varphi]$. If we have that $\beta(\mathcal{I})_i, \xi_i^k \Vdash_i ©_j[\varphi]$, then $last_i(\xi_i^k) \in E_j$ and $\beta(\mathcal{I})_i, last_i(\xi_i^k) \downarrow j \Vdash_j \varphi$. Using the induction hypothesis, we obtain $\mathcal{I}, [x/last_j(last_i(\xi_i^k) \downarrow j)] \Vdash_{FOL} f_j(\varphi, x)$. Additionally, $last_j(last_i(\xi_i^k) \downarrow j) = last_i(\xi_i^k) \in E_j$, so $@(j, last_i(\xi_i^k)) \in \mathcal{P}^{\mathcal{I}}$, which means that $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} @(j, x)$. From these results, we have that $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} @(j, x) \wedge f_j(\varphi, x)$, and thus, we arrive to the conclusion that $\mathcal{I}, [x/last_i(\xi_i^k)] \Vdash_{FOL} f_i(©_j[\varphi])$. Once again, the converse for this case is similar.

□

We are finally ready to prove the results that allow us to reach the most important result in this chapter: the fact that our translation function f, that translates DTL formulas into FOL formulas, preserves entailment. Again, we only regard in our translation FOL interpretation structures that satisfy $\{\bigwedge_{i \in Id} @(i, 0)\}$.

With this in mind, we introduce the two following propositions.

Proposition 2.2. Let $\Gamma \cup \{\delta\} \subseteq \mathcal{L}_{DTL}$. We have that

$$\text{if } \Gamma \models_{DTL} \delta \text{ then } f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\} \models_{FOL} f(\delta).$$

Proof. We start by proving that given a FOL interpretation structure $\mathcal{I}$, we have that, for every $\gamma \in \mathcal{L}_{DTL}$, $\beta(\mathcal{I}) \Vdash_{DTL} \gamma$ if and only if $\mathcal{I} \Vdash_{FOL} f(\gamma)$.

Like we did for the last proof, we assume that $last_i(\emptyset) = 0$. Let $\beta(\mathcal{I}) \nVdash @_i[\varphi]$. This means that there is a ξ_i^k such that $\beta(\mathcal{I})_i, \xi_i^k \nVdash_i \varphi$. By Proposition 2.1, we get that $\mathcal{I}, [x/last(\xi_i^k)] \nVdash_{FOL} f_i(\varphi, x)$. Additionally, $\mathcal{I}, [x/last(\xi_i^k)] \Vdash_{FOL} @(i, x)$. From this, we get that $\mathcal{I}, [x/last(\xi_i^k)] \nVdash_{FOL} @(i, x) \Rightarrow f_i(\varphi, x)$, meaning that $\mathcal{I} \nVdash_{FOL} \forall x\ (@(i, x) \Rightarrow f_i(\varphi, x))$. Thus $\mathcal{I} \nVdash_{FOL} f(@_i[\varphi])$.

Conversely, we assume that $\mathcal{I} \nVdash_{FOL} f(@_i[\varphi])$. This means that there is a $n \in \mathbb{N}_0$ such that $\mathcal{I}, [x/n] \nVdash_{FOL} @(i, x) \Rightarrow f_i(\varphi, x)$, i.e. $\mathcal{I}, [x/n] \Vdash_{FOL} @(i, x)$ and $\mathcal{I}, [x/n] \nVdash_{FOL} f_i(\varphi, x)$. From the first condition, we have that either $n = 0$, which leads to $(n \downarrow i) = \emptyset$, or $n \in E_i$, in which case it follows that $last_i(n \downarrow i) = n$. By Proposition 2.1, we obtain that $\beta(\mathcal{I})_i, n \downarrow i \nVdash_i \varphi$. Thus $\beta(\mathcal{I}) \nVdash_{DTL} @_i[\varphi]$.

We are now ready to finish our proof. Assume that $\Gamma \models_{DTL} \delta$ and let $\mathcal{I}$ be a FOL model that satisfies $f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\}$. From this, we get that $\beta(\mathcal{I}) \Vdash_{DTL} \Gamma$, and therefore $\beta(\mathcal{I}) \Vdash_{DTL} \delta$. Thus, $\mathcal{I} \Vdash_{FOL} f(\delta)$. From this reasoning, we reach the conclusion that $f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\} \models_{FOL} f(\delta)$. □

Proposition 2.3. Let $\Gamma \cup \{\delta\} \subseteq \mathcal{L}_{DTL}$. We have that

$$\text{if } f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\} \models_{FOL} f(\delta) \text{ then } \Gamma \models_{DTL} \delta.$$

Proof. We translate DTL interpretation structures into FOL interpretation structures. Given a DTL interpretation structure μ, we linearize its underlying global order of events $< Ev, \leq >$ by defining an injective function $h : Ev \to \mathbb{N}$ preserving the global causality relation, meaning that if $e < e'$ then $h(e) < h(e')$.

Thus, given a DTL interpretation structure μ and function h of $< Ev, \leq >$, we define an associated FOL interpretation structure $\mathcal{I}_{\mu,h}$ as having domain $\mathbb{N}_0$ and such that

$$p_i^{\mathcal{I}_{\mu,h}}(n) = \begin{cases} 1 & \text{if } (p \in \sigma_i(e \downarrow i) \wedge h(e) = n) \text{ or } (p \in \sigma_i(\emptyset) \wedge n = 0), \\ 0 & \text{otherwise,} \end{cases}$$

$$@^{\mathcal{I}_{\mu,h}}(i, n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } (e \in Ev_i \wedge h(e) = n), \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\mathcal{I}_{\mu,h}$ is a model of $(\bigwedge_{i \in Id} @(i, 0))$.

Now, by following a simple inductive argument similar to the one used in the proof of the previous proposition, we also get that, for every $\varphi \in \mathcal{L}_i$, $\mathcal{I}_{\mu,h}, [x/h(last_i(\xi_i))] \Vdash_{FOL} f_i(\varphi, x)$ if and only if $\mu_i, \xi_i \Vdash_i \varphi$. From this, we reach the conclusion that for every $\gamma \in \mathcal{L}_{DTL}$, $\mathcal{I}_{\mu,h} \Vdash_{FOL} f_i(\gamma)$ if and only if $\mu \Vdash_{DTL} \gamma$.

Finally we assume that $f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\} \models_{FOL} f(\delta)$. Also, let μ be a DTL model of Γ. Thus, $\mathcal{I}_{\mu,h} \Vdash_{FOL} f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\}$ meaning that $\mathcal{I}_{\mu,h} \Vdash_{FOL} f(\delta)$. We get $\mu \Vdash_{DTL} \delta$, allowing us to conclude that $\Gamma \models_{DTL} \delta$. □

The two previous propositions allow us to conclude the following result.

Corollary 2.1. Let $\Gamma \cup \{\delta\} \subseteq \mathcal{L}_{DTL}$. We have that

$$\Gamma \models_{DTL} \delta \text{ if and only if } f(\Gamma) \cup \{\bigwedge_{i \in Id} @(i, 0)\} \models_{FOL} f(\delta).$$

2.5 Extending the Translation Function to more Operators

Using this translation from DTL to FOL, one can prove (refute) the validity of a DTL formula by proving (refuting) the validity of the translation of the same formula into FOL. This DTL formula may include temporal operators whose direct translations we did not define, such as F, G and X, as it is enough to use the abbreviations of these operators in order to translate the DTL formula.

For a formula φ, the abbreviations of the operators F, G and X are the following:

- $\mathsf{F}\,\varphi \equiv \top\,\mathsf{U}\,\varphi$.
- $\mathsf{G}\,\varphi \equiv \neg\,\mathsf{F} \neg\,\varphi$.
- $\mathsf{X}\,\varphi \equiv \bot\,\mathsf{U}\,\varphi$.

However, in certain practical situations, it may be useful to consider the direct translations of these operators, since these are simpler than the abbreviations, as they contain, in fact, fewer quantifiers. For instance, if we consider a theorem prover capable of proving or refuting the validity of FOL formulas, this theorem prover will most likely be able to return a result faster if the direct translations of these operators are used.

Additionally, because of the added simplicity that comes with the use of direct translations, the theorem prover may be able to increase the range of formulas to which it can return a "valid" or "invalid" result. In chapter 3 of this document, we will use CVC4 to prove and refute the validity of LTL and DTL formulas by considering their translations into FOL (see also Appendix A). We confirmed that using the direct translations allowed for a wider range of formulas that we could prove or refute, due to the direct translations having a lower number of quantifiers.

With this in mind, we establish the lemmas that follow.

Lemma 2.1. The direct translation of the operator F from DTL into FOL is the following:

$$f_i(\mathsf{F}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi, y).$$

Proof. Note that the abbreviation of F is $\mathsf{F}\,\varphi \equiv \top\,\mathsf{U}\,\varphi$. From this, we have that $f_i(\top\,\mathsf{U}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi,y) \land \forall z\ (x < z < y \Rightarrow (@(i,z) \Rightarrow \top)))$. This leads to $f_i(\top\,\mathsf{U}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi,y) \land \forall z\ \top$, and simplifying, we get $f_i(\top\,\mathsf{U}\,\varphi, x) = \exists y\, x < y \land @(i,y) \land f_i(\varphi,y)$. □

Lemma 2.2. The direct translation of the operator G from DTL into FOL is the following:

$$f_i(\mathsf{G}\,\varphi, x) = \forall y\ (x < y \land @(i,y)) \Rightarrow f_i(\varphi, y).$$

Proof. Note that the abbreviation of G is $\mathsf{G}\,\varphi \equiv \neg\mathsf{F}\neg\varphi$. From this, we have that $f_i(\neg\mathsf{F}\neg\varphi, x) = \neg f_i(\mathsf{F}\neg\varphi, x) = \neg(\exists y\, x < y \land @(i,y) \land f_i(\neg\varphi, x))$. This means that $f_i(\neg\mathsf{F}\neg\varphi, x) = \forall y\, \neg(x < y \land @(i,y) \land f_i(\neg\varphi, x))$, and this leads to $f_i(\neg\mathsf{F}\neg\varphi, x) = \forall y\, \neg\neg((x < y \land @(i,y)) \Rightarrow \neg f_i(\neg\varphi, x))$, which simplifies to $f_i(\neg\mathsf{F}\neg\varphi, x) = \forall y\ (x < y \land @(i,y)) \Rightarrow f_i(\varphi, y)$. □

Lemma 2.3. The direct translation of the operator X from DTL into FOL is the following:

$$f_i(\mathsf{X}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi, y) \land \forall z\ (x < z < y \Rightarrow \neg @(i,z)).$$

Proof. The abbreviation of X is $\mathsf{X}\,\varphi \equiv \bot\,\mathsf{U}\,\varphi$. Thus, $f_i(\bot\,\mathsf{U}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi,y) \land \forall z(x < z < y \Rightarrow (@(i,z) \Rightarrow \bot))$. This can be simplified to $f_i(\bot\,\mathsf{U}\,\varphi, x) = \exists y\ x < y \land @(i,y) \land f_i(\varphi,y) \land \forall z(x < z < y \Rightarrow \neg @(i,z))$. □

Chapter 3

SMT Verification using CVC4

The SMT problem is a variant of the SAT problem for first-order logic, where the difference is in the fact that, for an SMT instance, the non-logical symbols are interpreted in the context of some background theory. Thus, the SMT problem consists of determining whether the SMT instance is satisfiable with respect to the background theory. Examples of theories typically used in computer science include the theory of integers, the theory of real numbers, the theory of lists or arrays and so on. Additionally, a given SMT instance might combine a set of theories (for instance, combining the theory of real numbers and the theory of integers).

CVC4 is an open-source theorem prover for SMT problems that can be used to prove the validity or the satisfiability of first-order formulas in several logical theories and their combination [14]. It was released in 2012 as the fourth in the CVC family of tools. It supports 4 different input languages, namely the CVC4 Native Input Language, SMT-LIB v2, SyGuS-IF and TPTP.

SMT problems have important applications in some areas of computer science, such as software verification, model checking and automated test generation [15]. Currently, CVC4 is considered to be one of the most efficient and up-to-date tools for solving these problems.

In this chapter, a summary of CVC4's functionalities is presented, along with its operability. Additionally, the methods used by CVC4 for handling satisfiability modulo theories (SMT) formulas are also discussed, with the focus being on the approach used specifically for SMT formulas with quantifiers. Lastly, we try to prove or refute the validity of LTL and DTL formulas using two different Mathematica functions we implemented, which translate LTL or DTL formulas into FOL formulas written in CVC4's native language. The code for these two Mathematica functions, and some indications on how to use them, can be found on Appendix A. Throughout this chapter, we use version 1.7 of CVC4. Furthermore, note that we do not show proofs for some of the results about CVC4. These proofs can be found in [16].

3.1 The Functionalities and Operability of CVC4

In this section, we focus on providing a summary of the CVC4 Native Input Language documentation, highlighting the most important features and how to operate with them. With this in mind, we start by describing the SAT and SMT problems, by first explaining what a literal is.

Definition 3.1. A literal is a either a propositional variable or the negation of a propositional variable. We say that x and y are positive literals, while $\neg x$ and $\neg y$ are said to be negative literals.

Definition 3.2. A clause is a disjunction of one or more literals.

An example of a clause is $x \vee \neg y \vee \neg z$.

Definition 3.3. A clausal formula is a conjunction of one or more clauses.

For instance, $(\neg x \vee y) \wedge (\neg y \vee w \vee \neg z)$ is a clausal formula.

The SAT problem is, then, the problem of determining whether a given clausal formula is satisfiable, i.e., if there exists an interpretation that satisfied a given clausal formula.

On the other hand, the SMT problem, also known as the satisfiability modulo theory problem, is a variant of the SAT problem for first-order logic such that, for an SMT instance, the variables are interpreted in the context of some background theory. Some theories that can be considered are, for example, the theory of integer numbers, real numbers and also theories of data structures, such as lists and arrays. Being an SMT-solver, CVC4 supports many of these theories.

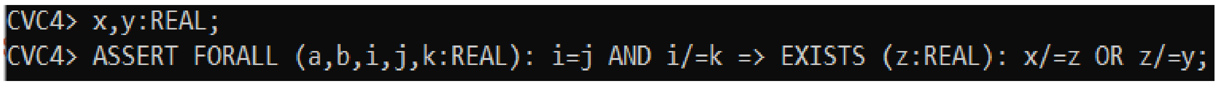
As mentioned in the beginning of the chapter, the SMT-solver CVC4 supports 4 different input languages, which differ in their syntax and in the names of certain commands. The language we used in this document is the CVC4 Native Input Language, whose documentation can be seen in [17]. It should also be mentioned that, even though the examples provided in this section were run in the (Windows) command line, CVC4 can also be run online [18].

CVC4 supports variables of different types, such as REAL, INT, BOOLEAN, STRING, array and tuple, to name a few. Variables can be declared in a similar way to some programming languages. Moreover, resorting to user-created variables or quantified variables, the ASSERT command can be used to add formulas to our current logical context. The current logical context Γ is a collection of the assertions the user has made so far, although CVC4 may also add formulas to the current context (this will be explained further ahead). CVC4 will always take into account the current logical context when executing queries.

As for the real and integer arithmetic theories, CVC4 currently supports numerals, along with the symbols $-$ (both unary and binary), $+$, $*$, $/$, $<$, $>$, $<=$ and $>=$. This excludes certain mathematical operations such as roots, powers, logarithms and factorials, for instance. Specific operations for other data types, like strings, arrays, sets or bit vectors, are also included in the tool.

In addition to these symbols, CVC4 allows the use of connectives such as $\forall$, $\Rightarrow$ and $\vee$, all of which have their own syntax. Figure 3.1 shows an example, run on the terminal, of the syntax in the declaration of variables, along with the ASSERT command, where some of the previously mentioned symbols and connectives are used.

CVC4's main functionalities come from the QUERY and CHECKSAT commands. The QUERY command gets a formula as input and checks if the formula is valid in the current logical context ($\Gamma \models F$, where Γ is our current logical context and F is the formula). This means that the QUERY command can be used to verify whether a given formula is a theorem or not. The QUERY command can produce 3 different answers:



```
CVC4> x,y:REAL;
CVC4> ASSERT FORALL (a,b,i,j,k:REAL): i=j AND i/=k => EXISTS (z:REAL): x/=z OR z/=y;
```

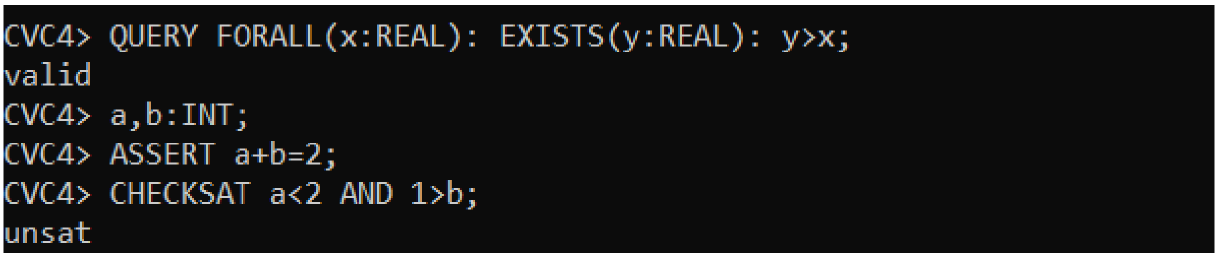
Figure 3.1: Example of the declaration of variables and the use of some symbols and connectives for the ASSERT command in CVC4 (run on the terminal).

- If the query returns “valid”, it means that $\Gamma \models_T F$. After this, the logical context stays exactly as it was before the query.

- If the query returns “invalid”, it means that $\Gamma \not\models_T F$. This implies that there is a model of the theory T that satisfies $\Gamma \cup \{\neg F\}$. After an invalid answer, the current logical context is augmented with a set Δ of variable-free literals such that $\Gamma \cup \Delta$ is satisfiable in T, but $\Gamma \cup \Delta \models_T \neg F$ (which in fact means that Δ entails $\neg F$). We call the new context $\Gamma \cup \Delta$ a counterexample for the formula F.

- If the query returns “unknown”, a set Δ of literals which entail $\neg F$ is added to the logical context, similarly to what occurs if the query returns “invalid”. However, the tool is not able to guarantee that $\Gamma \cup \Delta$ is satisfiable in T.

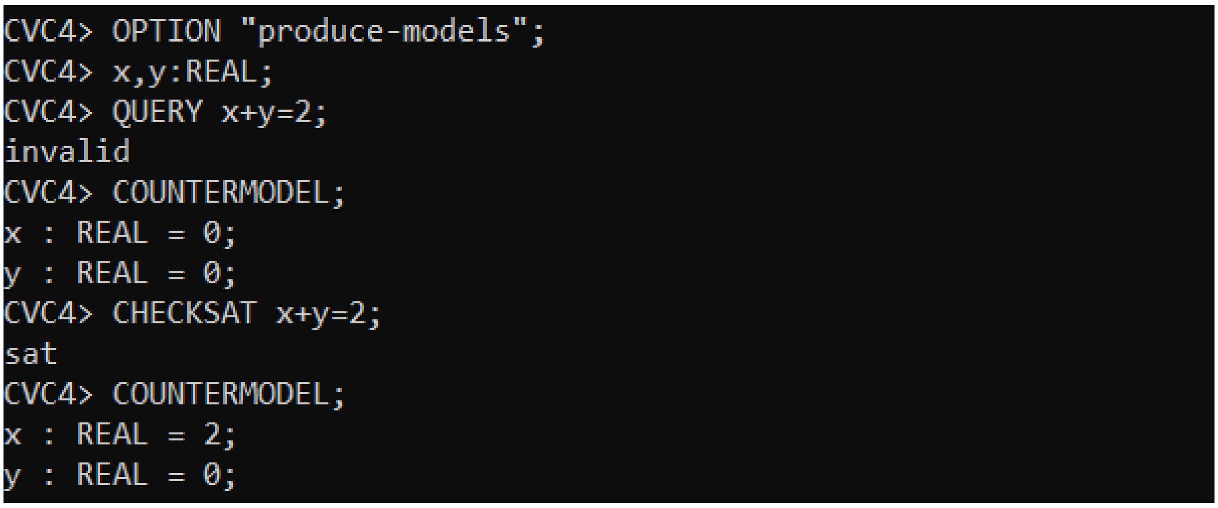
On the other hand, the CHECKSAT command also takes a formula as input and checks if the formula is satisfiable in the current logical context ($\Gamma \not\models \neg F$). Thus, this command behaves in the same way as making a query to $\neg F$, returning “sat” if $\neg F$ is invalid, “unsat” if $\neg F$ is valid, and “unknown” in the remaining cases. This command can be used to solve SAT and SMT instances.



```
CVC4> QUERY FORALL(x:REAL): EXISTS(y:REAL): y>x;
valid
CVC4> a,b:INT;
CVC4> ASSERT a+b=2;
CVC4> CHECKSAT a<2 AND 1>b;
unsat
```

Figure 3.2: Example of the QUERY command being used in CVC4 to prove a theorem and the CHECKSAT command being used to solve a SMT instance (run on the terminal).

Furthermore, if CVC4’s produce-models option is turned on (this can only be done on start-up, using the OPTION command), the user gains access to the COUNTERMODEL command. This command can be used to, after an invalid QUERY or a satisfiable CHECKSAT, print a model that makes the input formula invalid or satisfiable, respectively. An example of the application of these commands can be seen in Figure 3.3.



```
CVC4> OPTION "produce-models";
CVC4> x,y:REAL;
CVC4> QUERY x+y=2;
invalid
CVC4> COUNTERMODEL;
x : REAL = 0;
y : REAL = 0;
CVC4> CHECKSAT x+y=2;
sat
CVC4> COUNTERMODEL;
x : REAL = 2;
y : REAL = 0;
```

Figure 3.3: Example of the COUNTERMODEL command being applied in CVC4 to get a model that makes the formula $x + y = 2$ invalid after an invalid QUERY, and a model that makes the same formula satisfiable after a satisfiable CHECKSAT (run on the terminal).

CVC4 also contains other commands that can be advantageous in certain situations:

- The PUSH command saves the current state of the system, while POP restores the system to the state it was in right before the last PUSH. It can be useful if we want to return to the logical context we had before a possibly invalid QUERY or satisfiable CHECKSAT, since these results lead to a change in the context.
- The WHERE command prints all the formulas belonging to the current logical context.
- After an invalid QUERY or a satisfiable CHECKSAT, RESTART may be used to repeat the QUERY or CHECKSAT with an additional formula in the logical context. The formula needs to be introduced as input when calling the RESTART command.
- The TRANSFORM command takes a term as input, simplifies it using the current logical context and prints the result.
- PRINT and ECHO, which are common commands in programming languages, are also available.

Many modern SMT solvers, including CVC4, use a specific algorithm, called DPLL(T), to solve the SMT problem for quantifier-free SMT instances in an arbitrary theory T.

The DPLL(T) algorithm (also referred to as the lazy approach) transforms an SMT formula into an SAT one, by replacing every atom in the formula with Boolean variables [19]. Using the regular SAT-solving DPLL algorithm, a satisfying valuation for the new formula is found, if it exists (if not, the algorithm returns unsat). Afterwards, a theory solver is used to check if the assignments found are satisfiable in the theory T. We say that the theory solver checks if the assignments are T-satisfiable. If a contradiction is found by the theory solver, which means that the assignments are not T-satisfiable, then the algorithm refines the SAT formula with this information, and the regular DPLL algorithm is used once more on the SAT formula.

3.2 Quantifiers in a SMT Formula

The DPLL(T) algorithm can only be used for SMT formulas without quantifiers. For SMT formulas with quantifiers, a different approach is needed.

In [16], the authors describe the techniques used in CVC4 for solving linear arithmetic formulas with quantifiers. First, the article presents a framework that can be used to derive instantiation-based decision algorithms for formulas with one quantifier alternation. These algorithms can be used for any theory T, including in particular the linear real arithmetic (LRA) and linear integer arithmetic (LIA) theories. Furthermore, the article discusses the techniques used in CVC4 to extend these decision algorithms to formulas that have arbitrary quantifier alternations, and to mixed real and integer arithmetic.

For the following subsections, let us introduce some notation and definitions. A term written $t[\mathbf{k}]$ denotes a term whose free variables are in the tuple $\mathbf{k} = (k_1, ..., k_n)$. Similarly, a formula written $\varphi[\mathbf{j}]$ denotes a formula whose free variables are in the tuple $\mathbf{j} = (j_1, ..., j_m)$.

Given an integer $n > 1$, two integers a and b are said to be *congruent modulo* n, if n is a divisor of their difference (that is, if there is an integer k such that $a - b = kn$). We write $a \equiv b$ mod n.

Moreover, we say that a true/false decision problem is *decidable* if it can be solved by an algorithm that halts on all inputs in a finite number of steps.

3.2.1 An Instantiation Algorithm

In this subsection, let us assume a theory T and a language $\mathcal{E}$ such that $\mathcal{E}$ is a language closed under negation, and the satisfiability of finite sets of $\mathcal{E}$ formulas modulo T is decidable. We present an algorithm for checking the satisfiability of formulas in the language $\mathcal{Q}(\mathcal{E}) = \{\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}] \mid \varphi[\mathbf{k},\mathbf{x}] \in \mathcal{E}\}$. It should be noted that this procedure may be used for any theory T.

```
$\mathcal{P}_\mathcal{S}(\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}])$ :
    Let $\Gamma := \emptyset$ and $\mathbf{e}$ be a tuple of distinct fresh variables.
    Repeat
        If $\Gamma$ is $T$-unsatisfiable, then return “unsat”.
        If $\Gamma' = \Gamma \cup \{\neg\varphi[\mathbf{k},\mathbf{x}]\}$ is $T$-unsatisfiable, then return “sat”.
        Otherwise,
            Let $I$ be a model of $T$ and $\Gamma'$ and let $\mathbf{t}[\mathbf{k}] = \mathcal{S}(I, \Gamma, \neg\varphi[\mathbf{k},\mathbf{x}], \mathbf{e})$.
            $\Gamma := \Gamma \cup \{\varphi[\mathbf{k},\mathbf{t}[\mathbf{k}]]\}$
```

Algorithm 1: An instantiation-based algorithm $\mathcal{P}_\mathcal{S}$ for determining the T-satisfiability of $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$, using selection function $\mathcal{S}$.

As it was mentioned, the algorithm $\mathcal{P}_\mathcal{S}$ takes as input a quantified formula of the form $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$. It begins by considering a tuple of distinct fresh variables $\mathbf{e}$. Moreover, along the computation, it maintains a set of formulas Γ, which is initialized as the empty set. After initialization, a loop is entered; this loop will stop if either Γ is T-unsatisfiable, in which case it returns “unsat”, or $\Gamma \cup \{\neg\varphi[\mathbf{k},\mathbf{x}]\}$ is T-unsatisfiable, after which it returns “sat”. On each iteration of the loop, the algorithm invokes a function $\mathcal{S}$ (we call it a selection function), that returns a tuple of terms $\mathbf{t}[\mathbf{k}]$, whose free variables are a subset of $\mathbf{k}$. Afterwards, the formula $\varphi[\mathbf{k},\mathbf{t}[\mathbf{k}]]$ is added to Γ. We define selection function as follows:

Definition 3.4. A *selection function* (for $\mathcal{E}$) takes as arguments an interpretation I, a set of formulas Γ, and a formula $\neg\varphi[\mathbf{k},\mathbf{e}]$ in $\mathcal{E}$, and a tuple of variables $\mathbf{e}$, where $I \models \Gamma \cup \neg\varphi[\mathbf{k},\mathbf{e}]$. It returns a tuple of terms $\mathbf{t}[\mathbf{k}]$ such that $\varphi[\mathbf{k},\mathbf{t}[\mathbf{k}]]$ is also in $\mathcal{E}$.

By maintaining the set of formulas Γ, the algorithm tries to find a subset of instances of $\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$ that is either T-unsatisfiable, or T-satisfiable and entails $\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$. If the subset is T-unsatisfiable, this implies that $\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$ is also T-unsatisfiable. To check if the subset is T-satisfiable and entails $\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$, the algorithm checks on each iteration of the loop the satisfiability of $\Gamma \cup \neg\varphi[\mathbf{k},\mathbf{x}]$. If it is T-unsatisfiable, this necessarily means that $\neg\varphi[\mathbf{k},\mathbf{x}]$ is T-unsatisfiable, since the algorithm has checked in the previous step that Γ is T-satisfiable. In either case, the algorithm may terminate before all the instances of $\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$ are added to Γ. The correction of the procedure is a consequence of Lemma 3.1.

Lemma 3.1. If $\mathcal{P}_{\mathcal{S}}$ terminates when $\Gamma = \{\varphi[\mathbf{k},\mathbf{t}_1],\dots,\varphi[\mathbf{k},\mathbf{t}_n]\}$, then $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$ is equivalent to $\exists\mathbf{k}\varphi[\mathbf{k},\mathbf{t}_1] \wedge \dots \varphi[\mathbf{k},\mathbf{t}_n]$.

This lemma argues that the input to the procedure is equivalent to Γ, which means that the algorithm $\mathcal{P}_{\mathcal{S}}$ can effectively eliminate the quantifier $\forall$ on the input formula. Theorem 3.1 also provides us with a useful result regarding this algorithm.

Theorem 3.1. *If $\mathcal{S}$ is finite and monotonic for $\varphi[\mathbf{k},\mathbf{e}]$ in $\mathcal{E}$, then $\mathcal{P}_{\mathcal{S}}$ is a (terminating) decision procedure for the T-satisfiability of $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$.*

This theorem argues that the algorithm terminates as long as the selection function is finite and monotonic for the formula $\varphi[\mathbf{k},\mathbf{e}]$ in the language $\mathcal{E}$. The proofs for both Lemma 3.1 and Theorem 3.1 can be seen in [16].

3.2.2 A Selection Function for LIA-formulas

We consider formulas of the form $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$, where $\mathbf{k}$ and $\mathbf{x}$ are vectors of integer variables, and φ is quantifier-free. According to Theorem 1, to have a (terminating) decision procedure for the satisfiability of $\exists\mathbf{k}\forall\mathbf{x}\ \varphi[\mathbf{k},\mathbf{x}]$, we need to create a finite and monotonic selection function for LIA. The algorithm for creating such a function is described in this subsection. We assume that equalities are eliminated from φ using the following transformation:

$$t \cong 0 \rightsquigarrow 0 \leq t \wedge 0 \geq t.$$

We assume that the signature of integer arithmetic includes the symbols div^{+} and div^{-}, which denote division rounding up and down respectively. Additionally, we assume that LIA contains terms of the form

t div^p c, where $p \in \{+,-\}$ and c is a constant different from zero. We eliminate all occurrences of these terms from a quantifier-free formula using the transformation:

$$\varphi[t \text{ div}^p\ c] \rightsquigarrow \varphi[d] \wedge c \cdot d \cong t \pm^p m \wedge 0 \leq m < c,$$

where d and m are distinct fresh variables, and $\pm^p$ is $+$ if p is $+$ or $-$ if p is $-$.

The selection function $\mathcal{S}_{\mathsf{LIA}}$ is shown in Algorithm 2. The aim of this function is to find terms $\mathbf{t}$ to serve as substitutions for the variables $\mathbf{e}$ in $\neg\varphi[\mathbf{k},\mathbf{e}]$. From now on (and in the algorithm), we write $\psi\{e_i \mapsto t_i\}$ to denote the formula that results of substituting all occurrences of e_i by t_i in ψ. The algorithm calls the recursive algorithm $\mathcal{S}_r$, which takes as arguments I, $\neg\varphi[\mathbf{k},\mathbf{e}]$, variables $\mathbf{e}$ which we still did not incorporate into the substitutions, an integer θ, terms $\mathbf{t}$ which are the substitutions found for variables from $\mathbf{e}$ so far, and a tuple of symbols $\mathbf{p}$ (each one belonging to the set $\{+,-\}$) which we call polarities. The integer θ, which is initialized as 1, is meant to capture divisibility relationships through the algorithm.

```
$\mathcal{S}_{\mathsf{LIA}}(I, \Gamma, \neg\varphi[\mathbf{k},\mathbf{e}], \mathbf{e})$:
    Return $\mathcal{S}_r(I, \neg\varphi[\mathbf{k},\mathbf{e}], \mathbf{e}, 1, (), ())$.
$\mathcal{S}_r(I, \psi, (e_i, \ldots, e_n), \theta, \mathbf{t}, \mathbf{p})$:
    If $i > n$, return $\mathbf{t}$ div$^{\mathbf{p}}$ $\theta$
    Otherwise, let $(c, t_i, p_i) = \mathcal{S}_{sub}(I, \psi, e_i, \theta)$, $\sigma = \{c \cdot e_i \mapsto t_i\}$
    Return $\mathcal{S}_r(I, \psi\sigma, (e_{i+1}, \ldots, e_n), \theta \cdot c, ((c \cdot \mathbf{t})\sigma, \theta \cdot t_i), (\mathbf{p}, p_i))$
$\mathcal{S}_{sub}(I, \psi, e, \theta)$:
    Let $M = M_l \cup M_u \cup M_c$ be such that:
     — $I \models M$ and $M \models_p \psi$,
     —$M_l \Leftrightarrow \{c_1 \cdot e \geq l_1, \ldots, c_n \cdot e \geq l_n\},\ c_1 > 0, \ldots, c_n > 0$,
     —$M_u \Leftrightarrow \{d_1 \cdot e \leq u_1, \ldots, d_m \cdot e \leq u_m\},\ d_1 > 0, \ldots, d_m > 0$, and
     —$e \notin FV(l_1, \ldots, l_n) \cup FV(u_1, \ldots, u_m) \cup FV(M_c)$.
    Return one of:
```

$$\begin{cases} (c_i, l_i + \rho, +) & n > 0, \max\{(\frac{l_1}{c_1})^I, \ldots, (\frac{l_n}{c_n})^I\} = (\frac{l_i}{c_i})^I, \\ & \rho = (c_i \cdot e - l_i)^I \bmod (\theta \cdot c_i) \\ (d_j, u_j - \rho, -) & m > 0, \min\{(\frac{u_1}{d_1})^I, \ldots, (\frac{u_m}{d_m})^I\} = (\frac{u_j}{d_j})^I, \\ & \rho = (u_j - d_j \cdot e)^I \bmod (\theta \cdot d_j) \\ (1, \rho, +) & n = 0, m = 0, \rho = e^I \bmod \theta \end{cases}$$

Algorithm 2: A selection function $\mathcal{S}_{\mathsf{LIA}}$ for linear integer arithmetic LIA.

This procedure calls the (non-deterministic) algorithm $\mathcal{S}_{sub}(I, \psi, e_i, \theta)$ (where ψ is $\neg\varphi[\mathbf{k},\mathbf{e}]$ on the first call), which is meant to choose a term to substitute the variable e_i. The choice is made based on a set of literals M over the atoms of ψ which are satisfied by I and propositionally entail ψ. The literals in M are partitioned into three sets M_l, M_u and M_c, where the set M_l contains the literals that correspond to lower bounds for the variable e, while M_u contains the literals that correspond to upper bounds for the e, and M_c contains the literals that remain. Taking this into account, M_l is equivalent to a set that contains the literals in M_l written in solved form with respect to e, and the same is true for M_u. The literals in these sets are of the form $c_j \cdot e \geq l_j$ and $d_j \cdot e \leq u_j$, for M_l and M_u respectively. After considering these sets, the algorithm returns a tuple of the form (c, t_i, p_i), where c is a constant, t_i is a term, and p_i

is a polarity. This tuple is chosen non-deterministically, according to a set of conditions. If M_l is non-empty, the procedure may return the lower bound such that the value $(\frac{l_i}{c_i})^I$ is maximal, plus a constant ρ. Similarly, if M_u is non-empty, the procedure may return the upper bound such that the value $(\frac{u_i}{d_i})^I$ is minimal, minus a constant ρ. If both M_l and M_u are empty, the procedure returns the tuple $(1, \rho, +)$. The constant ρ is meant to guarantee that the returned term t_i and e are congruent modulo $\theta \cdot c$ in I, which is sufficient to show that $\mathcal{S}_{\mathsf{LIA}}$ is model-preserving.

Returning to $\mathcal{S}_r$, the algorithm constructs a substitution σ of the form $\{c \cdot e_i \mapsto t_i\}$. Subsequently, the recursive algorithm $\mathcal{S}_{sub}$ is called again, where σ is applied to ψ, θ is multiplied by c, $\theta \cdot t_i$ is appended to $(c \cdot \mathbf{t})\sigma$, and p_i is appended to $\mathbf{p}$.

After finding substitutions $\mathbf{t}$ for all the variables $\mathbf{e}$ in the original formula, the algorithm returns a vector of terms $\mathbf{t}$ div$^{\mathbf{p}}$ θ, which is the integer division of each term in $\mathbf{t}$ by θ, where the polarities in $\mathbf{p}$ determine if the division rounds up or down. When considering $\mathcal{P}_{\mathcal{S}_{\mathsf{LIA}}}$ (the instantiation procedure explained in subsection 3.2.1, parameterized by the selection function $\mathcal{S}_{\mathsf{LIA}}$), after the call to the selection function, the instance $\varphi[\mathbf{k}, \mathbf{t}$ div$^{\mathbf{p}}$ $\theta]$ will be added to Γ. Although the choice of polarities in the selection function does not have an impact in the correctness of the algorithm, it can reduce the number of instances needed in $\mathcal{P}_{\mathcal{S}_{\mathsf{LIA}}}$ for showing unsatisfiability.

Lemma 3.2. If I is a model for LIA and for quantifier-free ψ, $\mathcal{S}_{sub}(I, \psi, e, \theta) = (c, t, p)$, and $\theta \geq 1$, then:

1. $(c \cdot e)^I \equiv t^I$ mod $(\theta \cdot c)$, and

2. $I \models \psi\{c \cdot e \mapsto t\}$.

Lemma 3.3. Each recursive call to $\mathcal{S}_r(I, \psi, (e_i, \dots, e_n), \theta, (t_1, \dots, t_{i-1}), \mathbf{p})$, occurring within a call to $\mathcal{S}_{\mathsf{LIA}}(I, \Gamma, \neg\varphi[\mathbf{k}, \mathbf{e}], (e_1, \dots, e_n))$, is such that:

1. $\theta \mid t_j^I$ for each $1 \leq j < i$, and

2. $I \models \psi$ and ψ is equivalent to $\neg\varphi[\mathbf{k}, \mathbf{e}]\{\theta \cdot e_1 \mapsto t_1\} \cdot \ldots \cdot \{\theta \cdot e_{i-1} \mapsto t_{i-1}\}$.

Lemma 3.4. $\mathcal{S}_{\mathsf{LIA}}$ is model-preserving for $\varphi[\mathbf{k}, \mathbf{e}]$.

Theorem 3.2. *$\mathcal{P}_{\mathcal{S}_{LIA}}$ is a sound and complete procedure for determining the* LIA*-satisfiability of formulas of the form $\exists \mathbf{k} \forall \mathbf{x}\ \varphi[\mathbf{k}, \mathbf{x}]$.*

The proofs for the lemmas and the theorem in this subsection can be seen in [16].

We show some examples of the algorithm. In these examples, we chose terms $\mathbf{t}[\mathbf{k}]$ only based on the lower bounds M_l found in the algorithm $\mathcal{S}_{sub}$, although the algorithm may choose its terms based on the upper bounds M_u as well (recall that the procedure is non-deterministic). We also underlined the literal in M_l corresponding to the lower bound whose value is maximal according to I (which is the bound that is chosen by the algorithm).

Example 3.1. Let us consider the LIA-formula $\exists ab\ \forall xy\ (2 \cdot x < a \vee x + 3 \cdot y < b)$, whose negation (without considering the quantifiers and using a tuple of distinct fresh variables (e_1, e_2)) is $2 \cdot e_1 \geq a \wedge e_1 + 3 \cdot e_2 \geq b$. Table 3.1 summarizes a possible run of $\mathcal{P}_{\mathcal{S}_{LIA}}$.

					$\mathcal{S}_{sub}(I, \psi, e, \theta)$			
$\#$	Γ	Γ'	e	θ	M_l	return	$\mathbf{t}[\mathbf{k}]$	Add to Γ
1	sat	sat	e_1	1	$\{\underline{2 \cdot e_1 \geq a}, e_1 \geq b - 3 \cdot e_2\}$	$(2, a, +)$		
			e_2	2	$\{\underline{6 \cdot e_2 \geq 2 \cdot b - a}\}$	$(6, 2 \cdot b - a, +)$	$(6 \cdot a, 4 \cdot b - 2 \cdot a)\ div^+\ 12$	ψ_1
2	unsat							

Table 3.1: Possible run of $\mathcal{P}_{\mathcal{S}_{LIA}}$ for the input considered in Example 3.1.

For all calls to $\mathcal{S}_{sub}$ in this run, we assume $\rho = 0$. On the first call to $\mathcal{S}_{sub}$, we have that $e = e_1$, and the algorithm chooses the lower bound $2 \cdot e_1 \geq a$, returning the tuple $(2, a, +)$. The constant θ is multiplied by c, making it equal to 2, and t_1 is multiplied by 1, which was the value of θ used in this call to $\mathcal{S}_{sub}$. Afterwards, on the second call to $\mathcal{S}_{sub}$ (in which we have $e = e_2$), the algorithm applies the substitution $\{2 \cdot e_1 \mapsto a\}$, resulting in the bound $6 \cdot e_2 \geq 2 \cdot b - a$. This time, t_1 is multiplied by c, which is 6, and t_2 is multiplied by 2, which was the value of θ last used. With this, the algorithm $\mathcal{S}_{\mathsf{LIA}}$ returns the terms $(6 \cdot a, 4 \cdot b - 2 \cdot a)$ div$^+$ 12, and an instance, which we call ψ_1, is added to Γ in $\mathcal{P}_{\mathcal{S}_{\mathsf{LIA}}}$. This formula is equivalent to $2 \cdot ((6 \cdot a)$ div$^+$ $12) < a \vee (6 \cdot a)$ div$^+$ $12 + 3 \cdot ((4 \cdot b - 2 \cdot a)$ div$^+$ $12) < b$. After eliminating integer division, we obtain:

$$(2 \cdot k_1 < a \vee k_1 + 3 \cdot k_2 < b) \wedge 12 \cdot k_1 \cong 6 \cdot a + m_1 \wedge 0 \leq m_1 < 12 \wedge \\ 12 \cdot k_2 \cong (4 \cdot b - 2 \cdot a) + m_2 \wedge 0 \leq m_2 < 12$$

which is equisatisfiable to:

$$(6 \cdot a + m_1 < 6 \cdot a \vee 12 \cdot b + m_1 + 3 \cdot m_2 < 12 \cdot b) \wedge 0 \leq m_1 < 12 \wedge 0 \leq m_2 < 12$$

which is LIA-unsatisfiable. From this, the algorithm concludes that $\exists ab\ \forall xy\ (2 \cdot x < a \vee x + 3 \cdot y < b)$ is LIA-unsatisfiable.

Example 3.2. Let us consider the LIA-formula $\exists a\ \forall xy\ (3 \cdot x + y \not\cong a \vee 0 > y \vee y > 2)$, whose negation (without considering the quantifiers and using a tuple of distinct fresh variables (e_1, e_2)) is $3 \cdot e_1 + e_2 \cong a \wedge 0 \leq e_2 \wedge e_2 \leq 2$. This example demonstrates a case in which the value of ρ differs from zero. Table 3.2 summarizes a possible run of $\mathcal{P}_{\mathcal{S}_{\mathsf{LIA}}}$.

On the first iteration of $\mathcal{P}_{\mathcal{S}_{\mathsf{LIA}}}$, let us assume that I_1 is a model that satisfies Γ' and interprets all variables as zero. The algorithm $\mathcal{S}_{sub}$ chooses the maximal lower bounds $3 \cdot e_1 \geq a - e_2$ and $e_2 \geq 0$, and obtains $\rho = 0$, returning the tuples $(3, a - e_2, +)$ and $(1, 0, +)$. With this, the algorithm $\mathcal{S}_{\mathsf{LIA}}$ returns an instance ψ_1 that is added to Γ and is equivalent to $3 \cdot (a$ div$^+$ $3) \not\cong a$, which implies that $a^I \not\equiv 0$ mod 3 in subsequent models I that the algorithm chooses. This means that models I that satisfy $3 \cdot e_1 + e_2 \cong a$ are such that $e_2^I \not\equiv 0$ mod 3. On the next iteration of $\mathcal{P}_{\mathcal{S}_{LIA}}$, let I_2 be a model that satisfies Γ'. We obtain the same maximal lower bounds as before, meaning that, since I_2 satisfies $3 \cdot e_1 + e_2 \cong a$, we necessarily have that $\rho = ((e_2 - 0)^{I_2}$ mod $3) \neq 0$. The algorithm assumes that $(e_2 - 0)^{I_2} \equiv 1$ mod 3, thus assuming $\rho = 1$. The instance ψ_2 that is added to Γ' is equivalent to $3 \cdot ((a - 1)$ div$^+$ $3) + 1 \not\cong a$, implying that

			$\mathcal{S}_{sub}(I, \psi, e, \theta)$					
#	Γ	Γ'	e	θ	M_l	return	$\mathbf{t}[\mathbf{k}]$	Add to Γ
1	sat	sat	e_1	1	$\{\underline{3 \cdot e_1 \geq a - e_2}\}$	$(3, a - e_2, +)$		
			e_2	3	$\{\underline{e_2 \geq 0}\}$	$(1, 0, +)$	$(a, 0)$ div$^+$ 3	ψ_1
2	sat	sat	e_1	1	$\{\underline{3 \cdot e_1 \geq a - e_2}\}$	$(3, a - e_2, +)$		
			e_2	3	$\{\underline{e_2 \geq 0}\}$	$(1, 1, +)$	$(a - 1, 1)$ div$^+$ 3	ψ_2
3	sat	sat	e_1	1	$\{\underline{3 \cdot e_1 \geq a - e_2}\}$	$(3, a - e_2, +)$		
			e_2	3	$\{\underline{e_2 \geq 0}\}$	$(1, 2, +)$	$(a - 2, 2)$ div$^+$ 3	ψ_3
4	unsat							

Table 3.2: Possible run of $\mathcal{P}_{\mathcal{S}_{LIA}}$ for the input considered in Example 3.2.

$a^I \not\equiv 1$ mod 3 in subsequent models I. Thus, for the next models, we have that $e_2^I \not\equiv 1$ mod 3. On the third iteration, the algorithm uses $\rho = 2$, and so the instance ψ_3 is equivalent to $3 \cdot ((a-2)$ div$^+$ $3) + 2 \not\approx a$, implying that $a^I \not\equiv 2$ mod 3, which means that $e_2^I \not\equiv 2$ mod 3. Together, this instance and the two previous ones are LIA-unsatisfiable. Therefore, the algorithm concludes that $\exists a\ \forall xy\ (3 \cdot x + y \not\approx a \vee 0 > y \vee y > 2)$ is LIA-unsatisfiable.

3.2.3 The Generalized Algorithm

In this subsection, we present a technique that generalizes the instantiation-based algorithm of the previous subsection, making it capable of establishing the T-satisfiability of formulas with Boolean structure and nested quantification. First, we consider the following definition:

Definition 3.5. A first order logic formula is in *prenex normal form* if it is written as a string of quantifiers and bound variables, called the prefix, followed by a quantifier-free part, called the matrix.

We show an approach that can be used to establish the T-satisfiability of closed T-formulas $(\neg)\varphi$. Without loss of generality, let us consider that φ is a formula from the following grammar:

$$\varphi := \neg\forall x\ \varphi \mid G \mid \varphi_1 \vee \ldots \vee \varphi_m,$$

where G is quantifier-free. It is important to note that all formulas in prenex normal form are equivalent to a formula of this grammar for $m = 1$. Consequently, this grammar is a generalization of prenex normal form. Other approaches that are specialized for quantified SMT formulas often require that the input formula is in prenex normal form, yet the approach in this subsection may also be applied to inputs in the above grammar.

Fundamentally, our approach for establishing the T-satisfiability of φ initially consists in considering a sub-formula of φ of the form $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$, where ψ is quantifier-free. Note that this sub-formula may occur between any number of negations. Then, the instantiation-based algorithm of subsection 3.2.1 may be used to find a set of instances $\{\psi[\mathbf{k}, \mathbf{t}_1[\mathbf{k}]], \ldots, \psi[\mathbf{k}, \mathbf{t}_n[\mathbf{k}]]\}$ that is either T-unsatisfiable, or T-satisfiable and entails $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$. If the set is T-unsatisfiable, the algorithm replaces the sub-formula $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$ in φ by $\bot$, otherwise the algorithm replaces $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$ in φ by $\psi[\mathbf{k}, \mathbf{t}_1[\mathbf{k}]] \wedge \ldots \wedge \psi[\mathbf{k}, \mathbf{t}_n[\mathbf{k}]]$, which is quantifier-free. This process is repeated for every sub-formula until the original formula φ is replaced by a quantifier-free formula, to which the DPLL(T) algorithm may be applied afterwards. The algorithm we present in this subsection is based on the above reasoning, since it focuses on constructing a set of quantifier-free formulas Γ that replaces the input formula φ, while regularly checking the T-satisfiability of Γ.

We introduce some new concepts. For each closed quantified T-formula of the form $\forall \mathbf{x}\ \varphi$, let us consider a pair $(A, \mathbf{e})$, where $\mathbf{e}$ is a unique set of Skolem variables, and A is a Boolean variable called the *positive guard* of $\forall \mathbf{x}\ \varphi$. We write $(A, \mathbf{e}) \rightleftharpoons \forall \mathbf{x}\ \varphi$ to denote that A and $\mathbf{e}$ are associated to $\forall \mathbf{x}\ \varphi$. We write $\lfloor \varphi \rfloor$ to denote the formula that results of replacing all closed quantified formulas (that do not occur beneath other quantifiers) in φ with their corresponding positive guard. We write $\mathcal{A}(\varphi)$ to denote the set of positive guards in φ. The algorithm we present in this subsection maintains an evolving set of formulas Γ, and the formulas we add to Γ are of the form $A \Rightarrow \phi$, where ϕ is a quantifier-free formula entailed by $\forall \mathbf{x}\ \varphi$ and $(A, \mathbf{e}) \rightleftharpoons \forall \mathbf{x}\ \varphi$. We call such formulas *guarded instances*. We write $\lfloor \psi \rfloor_\Gamma$ to denote the result of replacing each positive guard A in $\lfloor \psi \rfloor$ by the conjunction of formulas on the right hand side of instances of Γ that are guarded by A (in other words, we select the instances where A is on the left hand side). The formula $\lfloor \psi \rfloor_\Gamma$ is meant to correspond to the quantifier-free approximation of ψ under the assumption of the current instances in Γ in our algorithm.

Example 3.3. Let φ be $\neg\forall x\ P(x) \vee \neg\forall y\ R(y) \vee \neg\forall z\ Q(z) \vee G$, where G is quantifier-free. Let $\forall x\ P(x) \rightleftharpoons (A_1, e_1)$, $\forall y\ R(y) \rightleftharpoons (A_2, e_2)$, $\forall z\ Q(z) \rightleftharpoons (A_3, e_3)$, and let Γ be $\{A_1 \Rightarrow P(a), A_2 \Rightarrow R(b), A_2 \Rightarrow R(c)\}$.

From this, we can conclude that $\lfloor \varphi \rfloor$ is $\neg A_1 \vee \neg A_2 \vee \neg A_3 \vee G$, $\mathcal{A}(\lfloor \varphi \rfloor) = \{A_1, A_2, A_3\}$, and $\lfloor \varphi \rfloor_\Gamma$ is $\neg P(a) \vee \neg(R(b) \wedge R(c)) \vee \neg\top \vee G$.

The technique used to establish the T-satisfiability of a closed T-formula $\forall \mathbf{x}\ \varphi[\mathbf{x}]$ is shown in Algorithm 3.

The algorithm $solve_T$ takes as input the closed T-formula and calls the subprocedure $CEGQI_T$, which in turn takes a set of formulas Γ (initially containing only the positive guard A_0 of $\forall \mathbf{x}\ \varphi[\mathbf{x}]$), and A_0 itself. This subprocedure maintains the set Γ, adding formulas to it using the recursive algorithm rec_T until either Γ is unsatisfiable, in which case the input closed T-formula is unsatisfiable, or the recursive procedure returns the empty set, which entails that the input closed T-formula is satisfiable.

Since our grammar has a tree-like structure, our algorithm uses a recursive procedure. This procedure, rec_T, is meant to go through the tree in order to find out if there exists a quantified sub-formula in the tree whose satisfiability is still unknown, in which case it returns a guarded instance of that sub-formula, using a selection function $\mathcal{S}_T$ to create that instance.

In more detail, the recursive algorithm rec_T takes as arguments the set of formulas Γ and the positive guard A of a formula $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$ (we have $A = A_0$ the first time rec_T is called). Then, the algorithm creates the formula $\phi[\mathbf{k}, \mathbf{e}] = \lfloor \psi[\mathbf{k}, \mathbf{e}] \rfloor_\Gamma$, which represents the approximation of $\psi[\mathbf{k}, \mathbf{e}]$ under the assumption of the current instances in Γ. If $\lfloor A \rfloor_\Gamma$ (the set of instances in Γ that are currently guarded by A) and $\neg\psi[\mathbf{k}, \mathbf{e}]$ are T-unsatisfiable, the algorithm returns the empty set. Else, the algorithm considers the

```
$solve_T(\forall \mathbf{x}\ \varphi[\mathbf{x}])$ :
    Return $CEGQI_T(\{\lfloor \forall \mathbf{x}\ \varphi[\mathbf{x}] \rfloor\}, \lfloor \forall \mathbf{x}\ \varphi[\mathbf{x}] \rfloor)$

$CEGQI_T(\Gamma, A_0)$ :
  1. If $\Gamma$ is $T$-unsatisfiable, then return "unsat".
  2. If $rec_T(\Gamma, A_0) = \emptyset$, then return "sat".
  3. Otherwise, return $CEGQI_T(\Gamma \cup rec_T(\Gamma, A_0), A_0)$.

$rec_T(\Gamma, A)$, where $(A, \mathbf{e}) \rightleftharpoons \forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$ :
   Let $\phi[\mathbf{k}, \mathbf{e}] = \lfloor \psi[\mathbf{k}, \mathbf{e}] \rfloor_\Gamma$.
   If $\lfloor A \rfloor_\Gamma \wedge \neg\phi[\mathbf{k}, \mathbf{e}]$ is $T$-unsatisfiable, then return $\emptyset$.
   If there exists an $A' \in \mathcal{A}(\lfloor \psi[\mathbf{k}, \mathbf{e}] \rfloor)$ such that $rec_T(\Gamma, A') \neq \emptyset$, then
         return $rec_T(\Gamma, A')$.
   Otherwise,
      Let $I$ be a model of $T$ and $\lfloor A \rfloor_\Gamma \wedge \neg\phi[\mathbf{k}, \mathbf{e}]$, and let
         $\mathbf{t}[\mathbf{k}] = \mathcal{S}_T(I, \Gamma, \neg\phi[\mathbf{k}, \mathbf{e}], \mathbf{e})$.
      Return $\{A \Rightarrow \phi[\mathbf{k}, \mathbf{t}[\mathbf{k}]]\}$.
```

Algorithm 3: An algorithm $solve_T$ for establishing the T-satisfiability of $\forall \mathbf{x}\ \varphi(\mathbf{x})$, where φ is a formula from the previously described grammar. This algorithm calls a counterexample-guided approach for quantifier instantiation $CEGQI_T$, and it generalizes the algorithm explained in the previous subsection. Instances are added to Γ using a selection function $\mathcal{S}_T$ for theory T.

quantified sub-formulas of ψ whose guarded instance occurs in $\mathcal{A}(\lfloor \psi[\mathbf{k}, \mathbf{e}] \rfloor)$ (the direct children of ψ), and if the recursive call to rec_T returns a guarded instance for any of the children, the algorithm returns that instance. Otherwise, the algorithm creates a model I of T and $\lfloor A \rfloor_\Gamma \wedge \neg\phi[\mathbf{k}, \mathbf{e}]$, and it returns a guarded instance $A \Rightarrow \phi[\mathbf{k}, \mathbf{t}[\mathbf{k}]]$, where the terms $\mathbf{t}[\mathbf{k}]$ are chosen by a selection function $\mathcal{S}_T$.

The correctness of $solve_T$ comes from the following results. For the next lemma, we say that two formulas are equivalent up to $\mathbf{k}$ if they are satisfied by the same set of models (when restricted to the interpretation of variables $\mathbf{k}$).

Lemma 3.5. Let $(A, \mathbf{e}) \rightleftharpoons \forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$.

1. If $rec_T(\Gamma, A)$ returns $\{A \Rightarrow \phi[\mathbf{k}, \mathbf{t}[\mathbf{k}]]\}$, then $\phi[\mathbf{k}, \mathbf{t}[\mathbf{k}]]$ is equivalent to $\psi[\mathbf{k}, \mathbf{t}[\mathbf{k}]]$ up to $\mathbf{k}$.

2. If $\lfloor A \rfloor_\Gamma \wedge \lfloor \neg\psi[\mathbf{k}, \mathbf{e}] \rfloor_\Gamma$ is T-unsat, then $\forall \mathbf{y}\ \psi[\mathbf{k}, \mathbf{y}]$ is equivalent to $\lfloor A \rfloor_\Gamma$ up to $\mathbf{k}$.

Theorem 3.3. *Assume the satisfiability of quantifier-free T-formulas is decidable, and a selection function $\mathcal{S}_T$ exists that is finite and monotonic.*

1. *If $solve_T(\forall \mathbf{x}\ \varphi[\mathbf{x}])$ returns "unsat", then $\forall \mathbf{x}\ \varphi[\mathbf{x}]$ is T-unsatisfiable.*

2. *If $solve_T(\forall \mathbf{x}\ \varphi[\mathbf{x}])$ returns "sat", then $\forall \mathbf{x}\ \varphi[\mathbf{x}]$ is T-satisfiable.*

3. *$solve_T(\forall \mathbf{x}\ \varphi[\mathbf{x}])$ terminates.*

Once again, the proofs for the lemma and theorem in this subsection can be seen in [16]. We show some examples of the algorithm.

Example 3.4. Let us consider the LIA-formula $\forall x\ \varphi[x]$, where $\varphi[x]$ is $\neg(\forall y\ x > y \vee 0 > y) \vee x < 0$. Let $(A_1, e_1) \rightleftharpoons \forall x\ \varphi[x]$ and let $(A_2, e_2) \rightleftharpoons \forall y\ e_1 > y \vee 0 > y$. The function $CEGQI$ is called, where Γ is initially $\{A_1\}$. Table 3.3 below summarizes a possible run of the algorithm.

		$rec_T(\Gamma, A)$						
#	Γ?	A	$\lfloor\neg\psi[\mathbf{e}]\rfloor$	$\lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$	$\lfloor A\rfloor_\Gamma \wedge \lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$?	$t[\mathbf{k}]$	return	return
1	sat	A_1	$A_2 \wedge e_1 \geq 0$	$\top \wedge e_1 \geq 0$	sat		$rec_T(\Gamma, A_2)$	
		A_2	$e_1 \leq e_2 \wedge 0 \leq e_2$	$e_1 \leq e_2 \wedge 0 \leq e_2$	sat	(e_1)	$\{A_2 \Rightarrow 0 > e_1\}$	
2	sat	A_1	$A_2 \wedge e_1 \geq 0$	$0 > e_1 \wedge e_1 \geq 0$	unsat		$\emptyset$	"sat"

Table 3.3: Possible run of the algorithm for the input considered in Example 3.4.

On the first call to $CEGQI$, Γ is T-satisfiable. The algorithm calls rec_T on Γ and A_1, which first checks if $\lfloor A_1\rfloor_\Gamma \wedge \lfloor A_2 \wedge e_1 \geq 0\rfloor_\Gamma$ is satisfiable. This is equivalent to $\top \wedge (\top \wedge e_1 \geq 0)$, which is satisfiable. Then, the algorithm checks if there is an A' among the positive guards in $\lfloor(\forall y\ e_1 > y \vee 0 > y) \wedge e_1 \geq 0\rfloor$ for which rec_T returns a guarded instance. For $A' = A_2$, we obtain that $e_1 \leq e_2 \wedge 0 \leq e_2$ is satisfiable. Since there are no positive guards in $\lfloor e_1 > e_2 \vee 0 > e_2\rfloor$ (because $\forall y\ e_1 > y \vee 0 > y$ contains no nested quantifiers), the algorithm uses the selection function $\mathcal{S}_{\mathsf{LIA}}$, which given input $e_1 \leq e_2 \wedge 0 \leq e_2$ returns the tuple (e_1), thus the instance $A_2 \Rightarrow 0 > e_1$ is added to Γ. On the second call to $CEGQI_T$, we get that $\Gamma = \{A_1, a_2 \Rightarrow 0 > e_1\}$ is satisfiable. The function rec_T is called on Γ and A_1, where we now get that $0 > e_1 \wedge e_1 \geq 0$ is unsatisfiable. From this, the algorithm concludes that $\forall x\ \neg(\forall y\ x > y \vee 0 > y) \vee x < 0$ is LIA-satisfiable.

Example 3.5. Let us consider the LIA-formula $\forall x\ \varphi[x]$, where $\varphi[x]$ is $(\neg(\forall y\ x > y) \vee \neg\forall z\ \psi[x])$, and $\psi[x]$ is some LIA-formula. Let $(A_1, e_1) \rightleftharpoons \forall x\ \varphi[x]$, let $(A_2, e_2) \rightleftharpoons \forall y\ e_1 > y$, and let $(A_3, e_3) \rightleftharpoons \forall z\ \psi[e_1]$. Table 3.4 summarizes a possible run of the algorithm.

		$rec_T(\Gamma, A)$						
#	Γ?	A	$\lfloor\neg\psi[\mathbf{e}]\rfloor$	$\lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$	$\lfloor A\rfloor_\Gamma \wedge \lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$?	$t[\mathbf{k}]$	return	return
1	sat	A_1	$A_2 \wedge A_3$	$\top \wedge \top$	sat		$rec_T(\Gamma, A_2)$	
		A_2	$e_1 \leq e_2$	$e_1 \leq e_2$	sat	(e_1)	$\{A_2 \Rightarrow e_1 > e_1\}$	
2	sat	A_1	$A_2 \wedge A_3$	$e_1 > e_1 \wedge \top$	unsat		$\emptyset$	"sat"

Table 3.4: Possible run of the algorithm for the input considered in Example 3.5.

On the first call to $CEGQI_T$, we find that Γ is satisfiable, thus rec_T is called. The recursive algorithm finds that $\lfloor A_1\rfloor_\Gamma \wedge \lfloor A_2 \wedge A_3\rfloor_\Gamma$ is T-satisfiable, meaning that rec_T is called on A_2, where the formula $A_2 \Rightarrow e_1 > e_1$ is added to Γ using the selection function $\mathcal{S}_{\mathsf{LIA}}$. On the second call to $CEGQI_T$, within the call to rec_T for $A = A_1$, the algorithm concludes that $\lfloor A_1\rfloor_\Gamma \wedge \lfloor(\forall y\ e_1 > y) \wedge \forall z\ \psi[e_1]\rfloor_\Gamma$, which is $\top \wedge (e_1 > e_1 \wedge \top)$ is unsatisfiable, meaning that rec_T returns the empty set and the algorithm returns "unsat". Thus, the formula $\forall x\ (\neg(\forall y\ x > y) \vee \neg\forall z\ \psi[x])$ is LIA-satisfiable.

Example 3.6. Let us consider the LIA-formula $\forall xy\ \varphi[x, y]$, where $\varphi[x, y]$ is $(\neg(\forall z z < x \vee y < z) \vee x < y + 5)$. Let $(A_1, (e_1, e_2)) \rightleftharpoons \forall xy\ \varphi[x, y]$ and let $(A_3, e_3) \rightleftharpoons \forall z\ z < e_1 \vee e_2 < z$. The algorithm $CEGQI$ is

		$rec_T(\Gamma, A)$						
#	Γ?	A	$\lfloor\neg\psi[\mathbf{e}]\rfloor$	$\lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$	$\lfloor A\rfloor_\Gamma \wedge \lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$?	$t[\mathbf{k}]$	return	return
1	sat	A_1	$A_3 \wedge e_1 \geq e_2 + 5$	$\top \wedge e_1 \geq e_2 + 5$	sat		$rec_T(\Gamma, A_3)$	
		A_3	$e_3 \geq e_1 \wedge e_2 \geq e_3$	$e_3 \geq e_1 \wedge e_2 \geq e_3$	sat	(e_2)	$\{A_3 \Rightarrow e_1 > e_2\}$	
2	sat	A_1	$A_3 \wedge e_1 \geq e_2 + 5$	$e_1 > e_2 \wedge e_1 \geq e_2 + 5$	sat		$\dots$	
		A_3	$e_3 \geq e_1 \wedge e_2 \geq e_3$	$e_3 \geq e_1 \wedge e_2 \geq e_3$	unsat		$\emptyset$	
		A_1	$\dots$	$\dots$	$\dots$	$(5, 0)$	$\{A_1 \Rightarrow \bot\}$	
3	unsat							“unsat”

Table 3.5: Possible run of the algorithm for the input considered in Example 3.6.

called, where Γ is initially $\{A_1\}$. Table 3.5 summarizes a possible run of $CEGQI$.

On the first call to $CEGQI$, Γ is satisfiable, and the recursive call to rec_T adds the instance $A_3 \Rightarrow e_1 > e_2$ to Γ. On the second call to $CEGQI$, Γ is again satisfiable, thus rec_T is called on A_1. It finds that $\lfloor A_1\rfloor_\Gamma \wedge \lfloor A_3 \wedge e_1 \geq e_2 + 5\rfloor_\Gamma$, which is $\top \wedge e_1 > e_2 \wedge e_1 \geq e_2 + 5$, is also satisfiable, and so rec_T is invoked recursively on A_3. On this call, the algorithm finds that $\lfloor A_3\rfloor_\Gamma \wedge \lfloor\neg\psi[\mathbf{e}]\rfloor_\Gamma$, which is $e_1 > e_2 \wedge (e_3 \geq e_1 \wedge e_2 \geq e_3)$, is unsatisfiable. The algorithm returns the empty set and comes back to rec_T for $A = A_1$. Lemma 5 implies that, since $e_1 > e_2 \wedge (e_3 \geq e_1 \wedge e_2 \geq e_3)$ is unsatisfiable, $\forall z\ z < e_1 \vee e_2 < z$ is equivalent to $e_1 > e_2$, which is the instance in Γ that is guarded by A_3.

Back on rec_T with $A = A_1$, the algorithm uses this information and replaces A_3 by $e_1 > e_2$ in the construction of $\lfloor\neg\psi[e_1, e_2]\rfloor_\Gamma$, which gives us $e_1 > e_2 \wedge e_1 \geq e_2 + 5$. This step is not explicitly written in the algorithm, but it occurs every time $rec_T(\Gamma, A')$ returns $\emptyset$ for some $A' \in \mathcal{A}(\lfloor\psi[\mathbf{k}, \mathbf{e}]\rfloor)$. Since there are no more positive guards in the original formula (we had only A_1 and A_2), the algorithm applies the selection function $\mathcal{S}_{\text{LIA}}$ to the new formula $e_1 > e_2 \wedge e_1 \geq e_2 + 5$, which returns the tuple $(5, 0)$ for (e_1, e_2). This gives us the instance $A_1 \Rightarrow \neg(5 > 0 \wedge 5 \geq 0 + 5)$, equivalent to $A_1 \Rightarrow \bot$, which is added to Γ. On the next call to $CEGQI$, the algorithm finds that Γ is unsatisfiable, returning “unsat”. Therefore, the formula $\forall xy\ (\neg(\forall z\ z < x \vee y < z) \vee x < y + 5)$ is LIA-unsatisfiable.

3.3 Validity Checking of LTL and DTL Formulas using CVC4

CVC4 does not directly support LTL, DTL, nor its operators. However, we have seen in chapter 2 that there is a translation from LTL into FOL, and we proved that a translation from DTL into FOL also exists. That is, any LTL or DTL formula can be transformed into a FOL formula through translation functions that preserve entailment.

Note that a FOL formula obtained using these translation functions will be such that every variable in it is of integer sort, and it can be a quantified formula (in the case of a translation from DTL into FOL, the translated formula will certainly be quantified). This means that, in truth, a translated LTL or DTL formula belongs to the LIA theory, like the formulas we have seen throughout this chapter. Thus, we should be able to use CVC4 to check the validity of LTL and DTL formulas, if we first translate them into FOL formulas written in an input language that CVC4 supports. Our aim in this section is to understand if the theorem prover is able to correctly check the validity of these formulas.

With this in mind, we implemented a Mathematica function that takes a LTL formula and the current moment in time as input, and returns as output the corresponding FOL formula, written in CVC4's native language. For this, we resorted to the translation function g, which is known to translate LTL formulas into FOL formulas, preserving entailment. We inductively define g the following way:

- $g(p, x) = p(x)$;
- $g(\neg\varphi, x) = \neg g(\varphi, x)$;
- $g(\varphi \Rightarrow \psi, x) = g(\varphi, x) \Rightarrow g(\psi, x)$;
- $g(\mathsf{X}\,\varphi, x) = g(\varphi, x + 1)$;
- $g(\mathsf{F}\,\varphi, x) = \exists y\ x < y \wedge g(\varphi, y)$;
- $g(\mathsf{G}\,\varphi, x) = \forall y\ x < y \Rightarrow g(\varphi, y)$;
- $g(\varphi\,\mathsf{U}\,\psi, x) = \exists y\ x < y \wedge g(\psi, y) \wedge \forall z\ (x < z < y \Rightarrow g(\varphi, z))$.

Formula	Expected output	CVC4's output	Counterexample
$\mathsf{X}\,(p \wedge q) \rightarrow \mathsf{X}\,p \wedge \mathsf{X}\,q$	valid	valid	
$\mathsf{X}\,p \wedge \mathsf{X}\,q \rightarrow \mathsf{X}\,(p \wedge q)$	valid	valid	
$\mathsf{X}\,(p \vee q) \rightarrow \mathsf{X}\,p \vee \mathsf{X}\,q$	valid	valid	
$\mathsf{X}\,p \vee \mathsf{X}\,q \rightarrow \mathsf{X}\,(p \vee q)$	valid	valid	
$\mathsf{F}\,(p \wedge q) \rightarrow \mathsf{F}\,p \wedge \mathsf{F}\,q$	valid	valid	
$\mathsf{F}\,p \wedge \mathsf{F}\,q \rightarrow \mathsf{F}\,(p \wedge q)$	invalid	unknown	
$\mathsf{G}\,(p \wedge q) \rightarrow \mathsf{G}\,p \wedge \mathsf{G}\,q$	valid	valid	
$\mathsf{G}\,p \wedge \mathsf{G}\,q \rightarrow \mathsf{G}\,(p \wedge q)$	valid	valid	
$\mathsf{F}\,(p \vee q) \rightarrow \mathsf{F}\,p \vee \mathsf{F}\,q$	valid	valid	
$\mathsf{F}\,p \vee \mathsf{F}\,q \rightarrow \mathsf{F}\,(p \vee q)$	valid	valid	
$\mathsf{G}\,(p \vee q) \rightarrow \mathsf{G}\,p \vee \mathsf{G}\,q$	invalid	unknown	
$\mathsf{G}\,p \vee \mathsf{G}\,q \rightarrow \mathsf{G}\,(p \vee q)$	valid	valid	
$\mathsf{F}\,p \rightarrow p$	invalid	invalid	$p = \lambda x.x == 1$ [1]
$\mathsf{F}\,\mathsf{F}\,p \rightarrow \mathsf{F}\,p$	valid	valid	
$\mathsf{G}\,\mathsf{G}\,p \rightarrow \mathsf{G}\,p$	valid	unknown	
$\mathsf{X}\,\mathsf{X}\,p \rightarrow \mathsf{X}\,p$	invalid	invalid	$p = \lambda x.x == 2$ [1]
$\mathsf{G}\,(p \rightarrow \mathsf{X}\,(p)) \rightarrow (p \rightarrow \mathsf{G}\,p)$	valid	–	
$\mathsf{G}\,(p \rightarrow \neg q \wedge \mathsf{X}\,p) \rightarrow (p \rightarrow \neg(r\,\mathsf{U}\,q)$	valid	–	

Table 3.6: Output of the QUERY command for LTL formulas.

[1] For this formula, the current moment in time was considered to be the instant 0.

The code for this Mathematica function can be seen in Appendix A.

After its implementation, we used the Mathematica function to convert various LTL formulas into FOL formulas written in CVC4's native language, and used the QUERY command to check the validity of each formula.

Table 3.6 shows the output of the QUERY command for these formulas. For the last two formulas in the table, the queries for both formulas did not halt, and therefore no output was given by CVC4.

It is possible to see that CVC4 was able to verify and refute the validity of many of the formulas. Overall, the results were positive, since although it has its flaws, CVC4 can be used to verify or refute the validity of LTL formulas. However, for some of the formulas, it was not able to reach a conclusion when it comes to their validity, therefore returning "unknown". Furthermore, for the more complex formulas (the last two formulas), the queries did not halt, potentially because CVC4 entered some sort of loop during both queries.

Similarly, a Mathematica function that takes a global DTL formula as input, and returns as output the corresponding FOL formula written in CVC4's native language, was implemented. For this, we resorted to the translation function defined in chapter 2. Again, the code for this function can be found in Appendix A. The Mathematica function was also applied on certain formulas, and then the QUERY command was used to check their validity.

First, the formula $@_i[\mathsf{G}\,(p \wedge q) \Rightarrow \mathsf{G}\,p \wedge \mathsf{G}\,q]$ was analyzed, among other DTL formulas consisting of a single global formula containing a local formula equal to a valid LTL formula shown on Table 3.6. The results were similar to the ones we had obtained for the LTL formulas, meaning that most of the DTL formulas were correctly determined to be valid.

Considering a different formula, CVC4 correctly established $@_i[\mathsf{F}\,p \Rightarrow p]$ as invalid, for which the command COUNTERMODEL provided the following counterexample:

$$@(id,k) = True \text{ and } p(id,k) = \begin{cases} k == 1 & \text{if } id = i \\ False & \text{otherwise.} \end{cases}$$

As for the formula $(@_i[\mathsf{G}\,(p \Rightarrow ©_j[q])] \wedge @_i[\mathsf{F}\,p]) \Rightarrow @_j[\mathsf{F}\,q]$, which should be valid, CVC4 returned "unknown", meaning that it was not able to reach a conclusion regarding the validity of the formula. We had hoped that CVC4 would be able to reach a successful output in this case. However, based on experiments we made with other formulas, CVC4 seems to have difficulty proving that DTL formulas containing communication formulas are valid, most likely because these formulas involve multiple agents and, therefore, more quantifiers, making them more complex than most of the other formulas we have shown.

Due to this, the results obtained for DTL formulas were not as good as we had hoped. Communication formulas are an important part of DTL, since they allow communication between different agents. Unfortunately, the fact that CVC4 has difficulty proving the validity of DTL formulas with communication makes it quite limited when it comes to distributed temporal logic.

Chapter 4

Separation Properties

In this chapter we introduce the separation property [4–7]. Intuitively, a logic has the separation property if every formula can be written as a Boolean combination of formulas that each only talk about the past, present and future.

A temporal logic is expressively complete if for every first-order formula in this fragment, there is a temporal logic formula that has exactly the same models (and vice-versa). A temporal logic is expressively complete if and only if it has the separation property, provided that the temporal logic can express the F and P operators (although we have not yet defined the P operator, it is similar to F, standing for "sometime in the past"). This makes the separation property very useful when it comes to understanding how expressive a temporal logic is, i.e., the quantity and variety of ideas that the temporal logic is able to represent.

The separation property is proven to hold for the temporal logic with the Until and Since operators over the integers [8]. We propose an extension of this property to DTL, and we prove that this extension of the separation property holds for the distributed temporal logic for which the local languages have both the Until and Since operators.

4.1 Separation Property for Temporal Logic

We start by introducing $\mathcal{L}_{TL}$, the language of the temporal logic with both Until and Since. Let us consider a countable set of propositional variables $Prop$.

Definition 4.1. The language of temporal logic $\mathcal{L}_{TL}$ is defined as follows:

$$\mathcal{L}_{TL} ::= Prop \mid \bot \mid \mathcal{L}_{TL} \Rightarrow \mathcal{L}_{TL} \mid \mathcal{L}_{TL} \,\mathsf{U}\, \mathcal{L}_{TL} \mid \mathcal{L}_{TL} \,\mathsf{S}\, \mathcal{L}_{TL}.$$

Note that the set $\mathcal{L}_{LTL}$ of LTL formulas can be defined just like the one of TL, but omitting the S case.

We now introduce certain concepts that are useful to understand the notion of separation.

Definition 4.2. We say a formula A is $simple$ if it has no outer Boolean structure, that is, if it is of the form p, $B \,\mathsf{S}\, C$, or $B \,\mathsf{U}\, C$ (for some $p \in Prop$ and $B, C \in \mathcal{L}_{TL}$).

Definition 4.3. A formula is called *non-future* if it has no occurrences of U and *non-past* if it has no occurrences of S.

A *pure past* formula is then a Boolean combination of formulas of the form $A\,\mathsf{S}\,B$ where both A and B are non-future and similarly a formula is *pure future* if it is a Boolean combination of formulas of the form $A\,\mathsf{U}\,B$ with A and B non-past.

A formula is *pure present* if it is a Boolean combination of variables, $\perp$ and $\top$.

Definition 4.4. A formula is *separated* if it is a Boolean combination of pure past, pure future and pure present formulas.

As mentioned before, TL has the separation property over the integers, a result that was proven by Gabbay in [8]. When we say “over the integers”, we are referring to the domain over which the formulas in the logic are evaluated.

With these definitions, we can proceed to our adaptation of the separation property to distributed temporal logic.

4.2 Separation Property for DTL

To understand how we reached our adaptation of the separation property to distributed temporal logic, there are certain details we first need to take into account.

Note that our definition of separation property for DTL needs only to focus on the local languages of DTL. This is true seeing that any temporal operators in a DTL formula occur in the local languages. In fact, the global language in DTL only deals with global formulas and the relations of global formulas between different agents. Since the global language of a distributed temporal logic is not defined using temporal operators, we can direct our attention to the local languages.

Regarding the formulas in the local languages, we can see that these coincide with temporal formulas, when excluding communication formulas. In turn, communication formulas contain assertions inside them that belong to other local languages of the distributed temporal logic, meaning that they may contain temporal operators inside them or even other communication formulas.

Due to the fact that communication formulas are what differentiates formulas in the local DTL languages from TL formulas, it would be useful to find a way to look at communication formulas as propositional symbols when it comes to separation, which we might be able to achieve if we can separate the assertions inside communication formulas. By looking at communication formulas in local DTL formulas as propositional symbols, it might be possible to separate these local DTL formulas using the same techniques required to separate TL formulas.

Following this train of thought, we reached definition 4.5.

Definition 4.5. A distributed temporal logic is said to have the *separation property* if its local formulas, for every agent, can be equivalently rewritten as a boolean combination of formulas, each of which depends only on the past, present or future.

Now that we have a definition of separation property for DTL, it is of our interest to try and prove that this property holds for distributed temporal logic.

With this intention, let us introduce the concept of complexity of a local DTL formula, which we define the following way:

- $|\varphi| = 0$ for all $\varphi \in \mathcal{L}_i^{\not\copyright}$, with $i \in Id$, where $\mathcal{L}_i^{\not\copyright}$ denotes all purely temporal formulas of $\mathcal{L}_i$, that is, excluding communication formulas;
- $|\copyright_j[\varphi]| = 1 + |\varphi|$, with $\varphi \in \mathcal{L}_j$, $j \in Id$;
- $|\varphi| = \max(|\copyright^1|, |\copyright^2|, \ldots, |\copyright^m|)$, where $\copyright^1, \copyright^2, \ldots, \copyright^m$ represent the communication formulas in φ that are not inside other communication formulas, with $\varphi \in \mathcal{L}_i$, $i \in Id$.

The complexity of a local DTL formula will be necessary in the next proof. In order to better understand this concept, we provide a few examples:

- $|(p \Rightarrow q) \wedge r| = 0$.
- $|\copyright_j[p \vee q]| = 1 + |p \vee q| = 1$, with $j \in Id$.
- $|(t \wedge \copyright_j[p \vee q]) \Rightarrow \copyright_i[r \Rightarrow \copyright_k[s]]| = \max(|\copyright_j[p \wedge q]|, |\copyright_i[r \Rightarrow \copyright_k[s]]|) = \max(1, 2) = 2$, with $i, j, k \in Id$.

Now, we are ready to prove Proposition 4.1.

Proposition 4.1. Distributed temporal logic, as defined in section 2.3, has the separation property.

Proof. First, note that, excluding communication formulas, local DTL formulas coincide with TL formulas. Additionally, note that, when it comes to local DTL formulas, an assertion inside a communication formula (and its temporal operators) concerns a different agent to the agent for which the communication formula holds.

The proof follows by induction on the complexity of the local DTL formula.

We need to consider two base cases:

1. For the first base case, let us consider local DTL formulas that have complexity 0, that is, local formulas belonging to the languages $\mathcal{L}_i^{\not\copyright}$, with $i \in Id$. Since formulas in the languages $\mathcal{L}_i^{\not\copyright}$ coincide with TL formulas, we can see assertions in these languages as TL formulas.

   Each one of the languages $\mathcal{L}_i^{\not\copyright}$ has the Until and the Since operators and, therefore, has the separation property over the integers. Consequently, by using the same techniques used to separate TL formulas, we can equivalently rewrite every possible local DTL formula in the languages $\mathcal{L}_i^{\not\copyright}$ (that is, local DTL formulas with complexity 0) as a boolean combination of formulas, each of which depends only on the past, present or future.

2. For the second base case, we consider local DTL formulas with complexity 1. These formulas may have one or more communication formulas, but each one of these cannot contain other communication formulas inside them.

Therefore, for this case, note that the assertions inside the communication formulas belong to the languages $\mathcal{L}_i^{\not\copyright}$, with $i \in Id$. This means that we can see the assertions inside the communication formulas as TL formulas. Again, by using the same techniques used to separate TL formulas, we can also separate the assertions contained in the communication formulas.

Furthermore, taking into account that a communication event may occur at any point in time, it follows that we can see these communication formulas as propositional symbols when it comes to separation. Thus, let us consider, for each $i \in Id$, the set of local state propositions $Prop'_i = Prop_i \cup \{\copyright_j[\mathcal{L}_j^{\not\copyright}] : j \in Id\}$.

With these new sets of local state propositions, let us consider, for each $i \in Id$, the corresponding temporal logic with set of propositional symbols $Prop'_i$ and having exactly the same operators as the local language of the agent i except for the communication formulas, which are seen as propositional symbols.

Each one of these languages has the Until and the Since operators and, therefore, has the separation property over the integers. This means that, by using the same techniques used to separate the formulas in these languages, we can equivalently rewrite local DTL formulas with complexity 1 as a boolean combination of formulas, each of which depends only on the past, present or future.

For the induction step, we consider a local DTL formula $\varphi \in \mathcal{L}_i$, for some i, that has complexity $|\varphi| = n$. Recall that $|\varphi| = \max(|\copyright^1|, |\copyright^2|, \dots, |\copyright^m|)$, where $\copyright^1, \copyright^2, \dots, \copyright^m$ represent the communication formulas in φ that are not inside other communication formulas. Note that the assertions inside each one of the communication formulas $\copyright^k$ have at most complexity $n-1$, for $1 \leq k \leq m$. Through the induction hypothesis, we know that the assertions inside the communication formulas $\copyright^k$ can be separated.

Thus, following the same train of thought as in the second base case, we can see each of the communication formulas $\copyright^k$ in the local DTL formula φ as propositional symbols when it comes to separation. Again, for $\mathcal{L}_i$, we consider the corresponding temporal logic with set of local state propositions containing also these communication formulas. The formulas in this language can be separated, and thus the same separation techniques for the formulas in this language can be used to separate φ.

Since every local formula in DTL, for every agent, can be separated, it follows that DTL has the separation property. $\square$

Example 4.1. Let φ be the DTL formula $@_j[(\copyright_k[c] \wedge (f \,\mathsf{U}\, g)) \,\mathsf{S}\, q] \Rightarrow @_i[\mathsf{F}\, d]$. We will use the algorithm that is implicit in the proof of Proposition 4.1 in order to find a separated formula equivalent to φ.

We will make use of a result for separating the LTL formula $\psi = (a \wedge (A \,\mathsf{U}\, B)) \,\mathsf{S}\, q$. This proof of this result and of other results for separating LTL formulas can be found in [8]. The result states that $\psi_1 \vee \psi_2 \vee \psi_3$ is a separated LTL formula equivalent to ψ, where

$$\psi_1 = (a \,\mathsf{S}\, q) \wedge (a \,\mathsf{S}\, B) \wedge B \wedge (a \,\mathsf{U}\, B);$$

$$\psi_2 = A \wedge (a \,\mathsf{S}\, (B \wedge q));$$

$$\psi_3 = (A \wedge q \wedge (a \,\mathsf{S}\, B) \wedge (a \,\mathsf{S}\, q)) \,\mathsf{S}\, q.$$

We look at the local formulas in φ. First, note that $\mathsf{F}\, d$ is an abbreviation of $\top \,\mathsf{U}\, d$, which is a formula that only talks about future. Therefore, this local formula requires no separation.

On the other hand, the local formula $(©_k[c] \wedge (f \,\mathsf{U}\, g)) \,\mathsf{S}\, q$ is clearly not separated. When it comes to complexity, $|(©_k[c] \wedge (f \,\mathsf{U}\, g)) \,\mathsf{S}\, q| = 1$, and so we have to follow the second base case of the proof of Proposition 4.1. There is only one communication formula, which is $©_k[c]$, and the assertion inside it is a single propositional symbol, thus it is already separated. Let $Prop_i$ be the set of propositional symbols of agent i. At this point, since the assertions inside the communication formulas in the local formula we are considering are already separated, we need to consider the temporal logic with set $Prop_i' = Prop_i \cup \{©_k[c]\}$. Then, due to the fact that we are now looking at $©_k[c]$ as if it was a propositional symbol, we are able to use the result presented in the beginning of this example in order to separate this local formula.

Hence, the separated DTL formula equivalent to φ is $(\phi_1 \vee \phi_2 \vee \phi_3) \Rightarrow @_i[\mathsf{F}d]$, where

$$\begin{aligned}
\phi_1 &= (©_k[c] \,\mathsf{S}\, q) \wedge (©_k[c] \,\mathsf{S}\, g) \wedge g \wedge (©_k[c] \,\mathsf{U}\, g);\\
\phi_2 &= f \wedge (©_k[c] \,\mathsf{S}\, (g \wedge q));\\
\phi_3 &= (f \wedge q \wedge (©_k[c] \,\mathsf{S}\, g) \wedge (©_k[c] \,\mathsf{S}\, q)) \,\mathsf{S}\, q.
\end{aligned}$$

Chapter 5

Craig Interpolation

In this chapter, we address the Craig interpolation property in the context of distributed temporal logic and we prove that a fragment of DTL has this property.

In classical logics, the Craig interpolation property states that if a formula ϕ entails a formula ψ, then there exists an interpolant formula θ such that ϕ entails θ, θ entails ψ, and every propositional symbol in θ occurs both in ϕ and ψ.

The Craig interpolation property is frequently a convenient property to have in a temporal logic. Temporal logics, in general, are widely used in the verification of systems and software, and interpolation has been useful for building efficient model-checkers. A stronger form of interpolation called uniform interpolation, which we will talk about in this chapter, has been particularly useful in this regard.

It has been proved that, in contrast to first-order logic, LTL does not have the Craig interpolation property. However, this property has been proved to hold for fragments of LTL, namely the fragment of LTL for which we consider X as the only temporal operator and exclude the rest of the temporal operators [11, 12]. Inspired by these proofs, we will show that the Craig interpolation property also holds for the fragment of DTL whose local languages contain X as the only temporal operator.

In this chapter, will often talk about fragments of DTL. We define fragments of the DTL by allowing in the syntax of their local languages only a subset of the temporal operators that the local languages of DTL are able to express.

5.1 Definition of Craig Interpolation in the Context of DTL

First, we need to introduce some concepts. Let us consider DTL (or a fragment of DTL) having global language $\mathcal{L}$ and local languages $\mathcal{L}_i$, for $i \in Id$. Additionally, let $\alpha = \{\alpha_i\}_{i \in Id}$ be a family of finite sets of propositional letters. We define $\mathcal{L}[\alpha]$ as the set of formulas in the global language $\mathcal{L}$ for which, for each $i \in Id$, the local formulas in the language $\mathcal{L}_i$ contain only propositional symbols from the set α_i.

Given an interpretation structure $\mu = \langle \lambda, \sigma \rangle = \langle \{\lambda_i\}_{i \in Id}, \{\sigma_i\}_{i \in Id} \rangle$ and a family of finite sets of propositional letters $\alpha = \{\alpha_i\}_{i \in Id}$, we define $\mu \upharpoonright \alpha$ as the interpretation structure with valuations $\{\sigma_i \upharpoonright \alpha_i\}_{i \in Id}$, where $\sigma_i \upharpoonright \alpha_i$ represents the restriction of the valuation σ_i to the propositional letters in α_i, for $i \in Id$.

We say that $\mu \restriction \alpha$ is the α-reduct of μ. We also say that μ is an expansion of $\mu \restriction \alpha$. Moreover, if $\mathcal{K}$ is a class of DTL interpretation structures, we write $\mathcal{K} \restriction \alpha$ for $\{\mu \restriction \alpha \mid \mu \in \mathcal{K}\}$.

Definition 5.1. Let $\mathcal{K}$ be a class of DTL interpretation structures and $\mathcal{L}$ be the global language of DTL (or of a fragment of DTL). We say that $\mathcal{K}$ is definable by $\mathcal{L}$ if there is a $\phi \in \mathcal{L}$ such that, for every interpretation structure μ, $\mu \Vdash \phi$ iff $\mu \in \mathcal{K}$.

Definition 5.2. Let $\mathcal{K}$ be a class of DTL interpretation structures and $\mathcal{L}$ be the global language of DTL (or a fragment of DTL) with set of agents Id. Additionally, let $\alpha = \{\alpha_i\}_{i \in Id}$ be a family of finite sets of propositional letters. Then $\mathcal{K}$ is *a projective class* of the global language $\mathcal{L}$ if there is a $\phi \in \mathcal{L}[\beta]$, with $\{\beta_i \supseteq \alpha_i\}_{i \in Id}$, such that $\mathcal{K} = Mod(\phi) \restriction \alpha$.

Now we define the Craig interpolation property for distributed temporal logic.

Definition 5.3. Let $\mathcal{L}$ be the global language of DTL (or a fragment of DTL) with set of agents Id, and $\alpha = \{\alpha_i\}_{i \in Id}$ and $\beta = \{\beta_i\}_{i \in Id}$ two families of finite sets of propositional letters. Then $\mathcal{L}$ has the *Craig interpolation property* whenever the following holds. Let $\phi \in \mathcal{L}[\alpha], \psi \in \mathcal{L}[\beta]$. Whenever $\phi \vDash \psi$, then there exists $\theta \in \mathcal{L}[\{\alpha_i \cap \beta_i\}_{i \in Id}]$ such that $\phi \vDash \theta$ and $\theta \vDash \psi$.

We also define uniform interpolation, a stronger form of interpolation than Craig interpolation.

Definition 5.4. Let $\mathcal{L}$ be the global language of DTL (or a fragment of DTL) with set of agents Id and let $\alpha = \{\alpha_i\}_{i \in Id}$ be a family of finite sets of propositional letters. Then $\mathcal{L}$ has *uniform interpolation* if, for all families of sets of propositional letters $\beta = \{\beta_i\}_{i \in Id}$ such that $\{\beta_i \subseteq \alpha_i\}$ for all $i \in Id$, and for each formula $\phi \in \mathcal{L}[\alpha]$, there is a formula $\theta \in \mathcal{L}[\beta]$ such that $\phi \vDash \theta$ and for each formula $\psi \in \mathcal{L}[\alpha']$ with $\{\alpha_i \cap \alpha'_i \subseteq \beta_i\}$ for all $i \in Id$, if $\phi \vDash \psi$ then $\theta \vDash \psi$.

To give an intuition, a fragment of DTL that has uniform interpolation is such that the interpolant can be constructed so that it depends only on the family of sets of propositional symbols of the antecedent and its intersection, for each agent, with the family of sets of propositional symbols of the consequent. The differences between Craig and uniform interpolation will be better explained in an example in the next section.

5.2 Craig Interpolation for a Fragment of DTL

Now, let us consider the fragment of DTL for which the local languages contain X as the only temporal operator (although they still contain communication formulas). We call this fragment DTL(X). The global language $\mathcal{L}$ of DTL(X) is defined by

$$\mathcal{L} ::= @_{i_1}[\mathcal{L}_{i_1}] \mid \cdots \mid @_{i_n}[\mathcal{L}_{i_n}] \mid \bot \mid \mathcal{L} \Rightarrow \mathcal{L},$$

for $Id = \{i_i, \ldots, i_n\}$, where the local languages $\mathcal{L}_i$ for each $i \in Id$ are defined by

$$\mathcal{L}_i ::= Prop_i \mid \bot \mid \mathcal{L}_i \Rightarrow \mathcal{L}_i \mid \mathsf{X}\, \mathcal{L}_i \mid \copyright_j[\mathcal{L}_j].$$

Theorem 5.1. *DTL(X) has uniform interpolation.*

Proof. We will show something stronger. In fact, we will show that every projective class of DTL(X) is definable by a formula in $\mathcal{L}$.

For this, let us consider a set of agents Id, an agent $j \in Id$ and a family $\alpha = \{\alpha_i\}_{i \in Id}$ of finite sets of propositional letters. Additionally, let $\phi \in \mathcal{L}[\{\alpha_i\}_{i \in Id \setminus \{j\}} \cup \{\alpha_j \cup p\}]$. Our proof will consist in showing how to construct a formula $\psi \in \mathcal{L}[\{\alpha_i\}_{i \in Id}]$ that defines a projective class of ϕ, that is, the class of α-reducts of models of ϕ.

With this in mind, let us focus on the agent j, since we aim to restrict the valuations σ_j. Let us consider the global formulas of agent j in ϕ, as well as the communication formulas of the form ©$_j[\varphi]$, for some $\varphi \in \mathcal{L}_j$, occurring in the global formulas in ϕ. Let m_1 be the depth of the maximal nesting of X-operators occurring in the global formulas of agent j in ϕ. Additionally, let c be the number of communication formulas of the form ©$_j[\varphi]$, for $\varphi \in \mathcal{L}_j$, occurring in ϕ. Let $m_2 = max(d_i + e_i)$, for $0 \leq i \leq c$, where d_i is the depth of the maximal nesting of X-operators occurring in communication formula i, and e_i represents the order of the local event in agent j at which communication formula i occurs (for example, if communication formula i occurs in the third local event of agent j, starting from the designated event, then $e_i = 3$). Finally, let n =max(m_1, m_2).

We can see that ϕ can only talk about n non-empty local events of agent j (starting from the designated event). Thus, it is possible to represent every valuation of p in these n local events by a set $S \subseteq \{0, \dots, n\}$, such that $k \in S$ means that p is true at the k-th event starting from the designated event, and false otherwise. Now, for each $S \subseteq \{1, \dots, n\}$, we define ϕ^S the following way:

- For each occurrence of p inside a global formula of agent j occurring in ϕ that is in the scope of k X-operators ($k \leq n$), we replace p by $\top$ if $k \in S$ and by $\bot$ otherwise;
- For each occurrence of p inside a communication formula of the form ©$_j[\varphi]$, for $\varphi \in \mathcal{L}_j$, occurring in ϕ, that synchronizes with agent j at its e-th local event, let k be the depth of the nesting of X-operators of this occurrence of p. We replace p by $\top$ if $(e + k) \in S$ and by $\bot$ otherwise.

We can show that ϕ and ϕ^S are equivalent in all interpretation structures for which the valuation of p is as is described by S.

For this, first note that the valuations for both formulas only differ in the global formulas of agent j and inside the communication formulas of the form ©$_j[\varphi]$, for $\varphi \in \mathcal{L}_j$. Thus, it is enough to show that $@[\varphi]$ and $@[\varphi^S]$ are equivalent, and that ©$_j[\omega]$ and ©$_j[\omega^S]$ are equivalent, in all interpretation structures for which the valuation of p is described by S, where $\varphi, \varphi^S, \omega, \omega^S \in \mathcal{L}_j$, and φ^S and ω^S are defined by replacing each occurrence of p as previously described.

Let us start with the global formulas of the form $@_j[\varphi]$ and $@_j[\varphi^S]$. We have to show that $\mu \Vdash @_j[\varphi]$ iff $\mu \Vdash @_j[\varphi^S]$, which is equivalent to showing that $\mu_j \Vdash_j \varphi$ iff $\mu_j \Vdash_j \varphi^S$. We show this by induction on the structure of the formulas.

For the base case, let φ be X...Xp, a nesting of k X-operators. If $\mu_j \Vdash_j \varphi$, then $\mu_j \Vdash_j$ X...Xp. Without loss of generality, if we fix the designated event, this means that p is true at the k-th event after the designated event, and therefore $k \in S$. Thus, φ^S is $\mathsf{X} \ldots \mathsf{X} \top$. We reach the conclusion that $\mu_j \Vdash_j \varphi^S$. For the reverse, let φ^S be $\mathsf{X} \ldots \mathsf{X} \top$, a nesting of k X-operators. If $\mu_j \Vdash_j \varphi^S$, then $\mu_j \Vdash_j \mathsf{X} \ldots \mathsf{X} \top$. If we fix the designated event, we can see that $k \in S$ (if this was not the case, p would not have been replaced by $\top$ in φ^S). Therefore $\mu_j \Vdash_j \varphi$, where φ is $\mathsf{X} \ldots \mathsf{X} p$.

For the step case, we consider two possibilities:

- Let φ be $\neg\varphi_1$, and φ^S be $\neg\varphi_1^S$. By the induction hypothesis, we have that $\mu_j \Vdash_j \varphi_1$ iff $\mu_j \Vdash_j \varphi_1^S$, which leads to $\mu_j \Vdash_j \neg\varphi_1$ iff $\mu_j \Vdash_j \neg\varphi_1^S$, which is what we wanted to prove.

- Let φ be $\varphi_1 \Rightarrow \varphi_2$, and φ^S be $\varphi_1^S \Rightarrow \varphi_2^S$. Let us assume that $\mu_j \Vdash_j \varphi_1 \Rightarrow \varphi_2$. Then either $\mu_j \Vdash_j \varphi_2$ or $\mu_j \nVdash_j \varphi_1$. If $\mu_j \Vdash_j \varphi_2$, then $\mu_j \Vdash_j \varphi_2^S$ by the induction hypothesis, and thus $\varphi_1^S \Rightarrow \varphi_2^S$. If $\mu_j \nVdash_j \varphi_1$, this means that $\mu_j \nVdash_j \varphi_1^S$ by the induction hypothesis, and this leads to $\varphi_1^S \Rightarrow \varphi_2^S$. The proof for the reverse is similar.

This finishes the proof that $@[\varphi]$ and $@[\varphi^S]$ are equivalent in the conditions previously mentioned.

Now, we need to show that $©_j[\omega]$ and $©_j[\omega^S]$ are also equivalent in all interpretation structures for which the valuation of p is as described by S. Note that we only need to focus on showing that $\mu_j \Vdash_j \varphi$ iff $\mu_j \Vdash_j \varphi^S$. From this, the rest of the proof can be done by induction on the structure of the formulas in a similar way to the proof we have just seen for the global formulas of agent j. We only need to take into account, for the base case, the event at which the communication formula synchronizes with agent j (recall how ϕ^S was defined for occurrences of p inside communication formulas).

Thus, we have shown that ϕ and ϕ^S are equivalent in all interpretation structures for which the valuation of p is as is described by S. Now, let $\psi = \bigvee_{S \subseteq \{0,\ldots,n\}} \phi^S$. We can conclude that ψ holds in an interpretation structure μ if and only if ϕ is satisfied by an expansion of μ. □

Corollary 5.1. DTL(X) has the Craig interpolation property.

Proof. Considering that uniform interpolation is a stronger form of interpolation than Craig interpolation, the result comes directly from Theorem 5.1. □

We show an example with the intent of better illustrating some of the concepts introduced in this chapter, including the differences between Craig and uniform interpolation.

Example 5.1. Let us consider DTL(X), which has both Craig interpolation and uniform interpolation. Let $\varphi \in \mathcal{L}[\alpha]$, where $\alpha = \{\alpha_j, \alpha_k\} = \{\{a,b,c\},\{p,q\}\}$. Also, let $\psi \in \mathcal{L}[\alpha']$, such that $\varphi \vDash \psi$, where $\alpha' = \{\alpha'_j, \alpha'_k\} = \{\{a\},\{p,q,r\}\}$. Let ϕ be the interpolant, such that $\phi \in \mathcal{L}[\beta]$, where $\beta = \{\beta_j, \beta_k\}$.

Note that both Craig and uniform interpolation require that $\varphi \vDash \phi$. For Craig interpolation, it is also necessary to consider the conditions $\phi \vDash \psi$ and $\phi \in \mathcal{L}[\{\alpha_i \cap \alpha'_i\}_{i \in Id}]$, that is, $\beta_j \subseteq \alpha_j \cap \alpha'_j$ and $\beta_k \subseteq \alpha_k \cap \alpha'_k$. We conclude that, for this type of interpolation, the interpolant depends on the antecedent, the consequent and the intersection of their sets of propositional symbols, for each agent, in the formulas φ and ψ. In this particular example, $\beta_j \subseteq \{a\}$ and $\beta_k \subseteq \{p,q\}$.

On the other hand, for uniform interpolation, the restrictions are the following: $\alpha_j \cap \alpha'_j \subseteq \beta_j$, $\alpha_k \cap \alpha'_k \subseteq \beta_k$, $\beta_j \subseteq \alpha_j$ and $\beta_k \subseteq \beta_k$. The interpolant depends on the antecedent and the intersection of its sets of propositional symbols, for each agent, with the sets of propositional symbols of the consequent. However, it does not depend on the consequent, making uniform interpolation stronger that Craig interpolation. From this, we have that $\beta_j \subseteq \{a, b, c\}$, $\beta_k \subseteq \{p, q\}$, $\{a\} \subseteq \beta_j$ and $\{p, q\} \subseteq \beta_k$.

Chapter 6

Conclusions

6.1 Achievements

One of the goals of this work was to show how to translate from DTL into FOL. With this in mind, we have presented a translation function that translates DTL formulas into FOL formulas, and we have shown that it preserves entailment in DTL.

We have also studied the theorem-prover CVC4, and following the work done in [16], we have presented the algorithms used in the theorem-prover for proving (or refuting) the validity of quantified formulas in the LIA theory. Since LTL and DTL formulas translated into FOL belong to the LIA theory, and typically contain quantifiers, we created two Mathematica functions, which can be seen in Appendix A, that translate LTL and DTL formulas into FOL formulas written in CVC4's native language. With this, we wanted to understand if it was possible to use CVC4 to check the validity of LTL and DTL formulas. While this method shows some positive results for LTL formulas, it fares worse when it comes to DTL formulas, specifically the ones that contain communication formulas. This likely occurs because these formulas involve multiple agents, which lead to more quantifiers on the translated formula, often making them more complex. Unfortunately, we are not able to recommend our approach with CVC4 for checking the validity of DTL formulas.

We have proposed an extension of the separation property to DTL, based on the definition of this property for temporal logic [4–8]. We have proved that this extension of the separation property holds for the distributed temporal logic for which the local languages have both the Until and Since operators.

Finally, we have addressed the Craig interpolation property in the context of DTL and we have proved that this property holds for the fragment of DTL for which the local languages contain X as the only temporal operator.

6.2 Future Work

Even though CVC4 is currently considered to be one of the most up-to-date theorem provers for SMT problems, it would be interesting to study another theorem prover and to understand if it could provide more consistent results in our approach for checking the validity of DTL formulas.

Bibliography

[1] D. Basin, C. Caleiro, J. Ramos, and L. Vigano. Distributed temporal logic for the analysis of security protocol models. *Theor. Comput. Sci.*, 412:4007–4043, July 2011. doi: 10.1016/j.tcs.2011.04.006. URL `https://doi.org/10.1016/j.tcs.2011.04.006`.

[2] D. Basin, C. Caleiro, J. Ramos, and L. Viganò. Labelled tableaux for distributed temporal logic. *Journal of Logic and Computation*, 19(6):1245–1279, January 2009. ISSN 0955-792X. doi: 10.1093/logcom/exp022.

[3] H.-D. Ehrich and C. Caleiro. Specifying communication in distributed information systems. *Acta Informatica*, 36:591–616, 03 2000. doi: 10.1007/s002360050167.

[4] D. Oliveira and J. Rasga. Revisiting separation: algorithms and complexity. *Logic Journal of the IGPL*, 02 2020. ISSN 1367-0751. doi: 10.1093/jigpal/jzz081. URL `https://doi.org/10.1093/jigpal/jzz081`. jzz081.

[5] A. Rabinovich. A proof of kamp's theorem. *Log. Methods Comput. Sci.*, 10(1), 2014. doi: 10.2168/LMCS-10(1:14)2014. URL `https://doi.org/10.2168/LMCS-10(1:14)2014`.

[6] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004. ISBN 3540002960.

[7] D. Oliveira. Linear Temporal Logic: Separation and Translation. 2017. Master's thesis, Instituto Superior Técnico.

[8] D. Gabbay. The declarative past and imperative future. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, pages 409–448, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. ISBN 978-3-540-46811-0.

[9] D. Peled. Temporal Logic: Mathematical Foundations and Computational Aspects, Volume 1. *The Computer Journal*, 38(3):260–261, 01 1995. ISSN 0010-4620. doi: 10.1093/comjnl/38.3.260. URL `https://doi.org/10.1093/comjnl/38.3.260`.

[10] I. Hodkinson and M. Reynolds. Separation - past, present, and future. In *We Will Show Them!*, 2005.

[11] A. Gheerbrant and B. ten Cate. Craig interpolation for linear temporal languages. In E. Grädel and R. Kahle, editors, *Computer Science Logic*, pages 287–301, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] N. Kamide. Interpolation theorems for some variants of ltl. *Reports on Mathematical Logic*, 2015 (Number 50), 2015. URL `https://www.ejournals.eu/rml/2015/Number-50/art/5718/`.

[13] G. De Giacomo and M. Vardi. Linear temporal logic and linear dynamic logic on finite traces. *IJCAI International Joint Conference on Artificial Intelligence*, pages 854–860, January 2013.

[14] About CVC4, 2020. URL `https://cvc4.github.io/index.html`. Last accessed 30 January 2020.

[15] About SMT-LIB, 2020. URL `http://smtlib.cs.uiowa.edu/about.shtml`. Last accessed 30 January 2020.

[16] A. Reynolds, T. King, and V. Kuncak. Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods in System Design*, 2017. doi: 10.1007/s10703-017-0290-y.

[17] CVC4 Native Input Language, 2020. URL `https://cvc4.github.io/cvc4-native-input-language.html`. Last accessed 04 February 2020.

[18] CVC4 Online App, 2020. URL `https://cvc4.github.io/app/`. Last accessed 04 February 2020.

[19] A. Reynolds. Satisfiability modulo theories and DPLL(T), 2015. URL `http://homepage.divms.uiowa.edu/~ajreynol/pres-dpllt15.pdf`. Last accessed 30 January 2020.

Appendix A

Code

In this appendix, we present the Mathematica functions created for translating LTL and DTL formulas into FOL formulas written in CVC4's native input language, in order to check their validity using CVC4.

Before presenting the code, we provide some pointers on how to use it, particularly on how to write in Mathematica the LTL and DTL formulas that serve as input for the functions. Although Mathematica contains operators such as $\neg$ and $\Rightarrow$, it does not have temporal operators or communication formulas, for instance. Because of this, it was necessary to find a way to write these operators in Mathematica, so that it would be possible to express LTL and DTL formulas.

For propositional symbols, a simple letter can be used. A string composed of a letter concatenated with an integer may be used as well.

As for the operators, Table A.1 explains the way to write them.

Operator	In Mathematica
$\neg\varphi$	Not[φ]
$\varphi \Rightarrow \psi$	Implies[φ,ψ]
$\varphi \wedge \psi$	And[φ,ψ]
$\varphi \vee \psi$	Or[φ,ψ]
X φ	X[φ]
F φ	F[φ]
G φ	G[φ]
φUψ	Until[φ,ψ]
$@_i[\varphi]$	At[i,φ]
φSψ	Since[φ,ψ]
©$_j[\varphi]$	Communication[j,φ]

Table A.1: Operators and the way to write them in Mathematica.

Now, we transcript the code developed.

Listing A.1: Function that translates LTL formulas into FOL formulas written in CVC4's native input language

```
TranslFuncLTLtoFOL =
  Function[{formula, time}, Module[{symbols, i, cvcstring, querytext},
    symbols = DeleteDuplicates@Cases[formula, _Symbol, Infinity];
    cvcstring = {};
    For[i = 1, i <= Length[symbols], i++,
     If [! (symbols[[i ]] === True || symbols[[i ]] === False),
      cvcstring =
        cvcstring <> ToString[symbols[[i]]] <>
         ":␣INT␣−>␣BOOLEAN;\n";]];
    querytext = aux[formula, time];

    cvcstring = cvcstring <> "QUERY␣" <> querytext <> ";"
     ]];

aux = Function[{formula, time}, Module[{varsuffix},
    Which[Head[formula] === Symbol,
     If [formula === True, "TRUE",
      If [formula === False, "FALSE",
       ToString[formula] <> "(" <> ToString[time] <> ")"]],
     Head[formula] === Implies,
     "(" <> aux[formula[[1]], time] <> ")␣=>␣(" <>
      aux[formula [[2]], time] <> ")",
     Head[formula] === Not, "NOT␣(" <> aux[formula[[1]], time] <> ")",
     Head[formula] === And,
     "(" <> aux[formula[[1]], time] <> ")␣AND␣(" <>
      aux[formula [[2]], time] <> ")",
     Head[formula] === Or,
     "( " <> aux[formula[[1]], time] <> ")␣OR␣(" <>
      aux[formula [[2]], time] <> ")",
     Head[formula] === X, aux[formula[[1]], ToString[time] <> "+1"],
     Head[formula] === F, varsuffix = RandomInteger[10^5];
     "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣(j" <>
      ToString[varsuffix] <> "␣>=␣" <> ToString[time] <> "␣AND␣(" <>
      aux[formula [[1]], "j" <> ToString[varsuffix]] <> "))",
     Head[formula] === G, varsuffix = RandomInteger[10^5];
     "FORALL(i" <> ToString[varsuffix] <> ":␣INT):␣i" <>
      ToString[varsuffix] <> "␣>=␣" <> ToString[time] <> "␣=>␣(" <>
      aux[formula [[1]], "i" <> ToString[varsuffix]] <> ")",
     Head[formula] === Until, varsuffix = RandomInteger[10^5];
     "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣j" <>
      ToString[varsuffix] <> ">=␣" <> ToString[time] <> "␣AND␣(" <>
      aux[formula [[2]], "j" <> ToString[varsuffix]] <>
      ")␣AND␣FORALL(k" <> ToString[varsuffix] <> ":␣INT):␣((" <>
      ToString[time] <> "␣<=␣k" <> ToString[varsuffix] <> ")␣AND␣(k" <>
       ToString[varsuffix] <> "␣<␣j" <> ToString[varsuffix] <>
      "))␣=>␣(" <> aux[formula[[1]], "k" <> ToString[varsuffix]] <>
      ")" ]]];
```

Listing A.2: Function that translates DTL formulas into FOL formulas written in CVC4's native input language

```
TranslFuncDTLtoFOL =
  Function[{formula}, Module[{symbols, i, cvcstring, communication},
    symbols = DeleteDuplicates@Cases[formula, _Symbol, Infinity];
    communication =
     DeleteDuplicates@Cases[formula, _Communication, Infinity];
    symbols = Rest[symbols];
    For[i = 1, i <= Length[communication], i++,
     If [MemberQ[symbols, communication[[i, 1]]],
      symbols = DeleteCases[symbols, communication[[i, 1]]]]];
    cvcstring = {};
    For[i = 1, i <= Length[symbols], i++,
     If [! (symbols[[i ]] === True || symbols[[i ]] === False),
      cvcstring =
       cvcstring <> ToString[symbols[[i]]] <>
        ":␣(STRING,␣INT)␣−>␣BOOLEAN;\n"]];
    cvcstring = cvcstring <> "At:␣(STRING,␣INT)␣−>␣BOOLEAN;\n";
    cvcstring = cvcstring <> "QUERY␣" <> auxglobal[formula] <> ";"
    ]];

auxglobal = Function[{formula}, Module[{varsuffix},
    Which[
     Head[formula] === Implies,
     "( " <> auxglobal[formula[[1]]] <> ")␣=>␣(" <>
      auxglobal[formula [[2]]] <> ")",
     Head[formula] === Not, "NOT␣(" <> auxglobal[formula[[1]]] <> ")",
      Head[formula] === And,
      "( " <> auxglobal[formula[[1]]] <> ")␣AND␣(" <>
       auxglobal[formula [[2]]]  <> ")",
      Head[formula] === Or,
      "( " <> auxglobal[formula[[1]]] <> ")␣OR␣(" <>
       auxglobal[formula [[2]]]  <> ")",
      Head[formula] === At,
      "FORALL(n:␣INT):␣n>=0␣=>␣(At(\""␣<>␣ToString[formula[[1]]]␣<>
␣␣␣␣␣␣"\",␣n)␣=>␣(" <> auxlocal[formula[[1]], formula [[2]],  "n"]  <>
       "))" ]]];

auxlocal = Function[{agent, formula, time}, Module[{varsuffix},
     Which[Head[formula] === Symbol,
      If [formula === True, "TRUE",
       If [formula === False, "FALSE",
        ToString[formula] <> "(\""␣<>␣ToString[agent]␣<>␣"\",␣" <>
         ToString[time] <> ")"]],
      Head[formula] === Implies,
      "( " <> auxlocal[agent, formula[[1]],  time]  <> ")␣=>␣(" <>
       auxlocal[agent, formula [[2]],  time]  <> ")",
      Head[formula] === Not,
      "NOT␣(" <> auxlocal[agent, formula[[1]], time]  <> ")",
      Head[formula] === And,
      "( " <> auxlocal[agent, formula[[1]],  time]  <> "␣AND␣" <>
       auxlocal[agent, formula [[2]],  time]  <> ")",
      Head[formula] === Or,
      "( " <> auxlocal[agent, formula[[1]],  time]  <> ")␣OR␣(" <>
       auxlocal[agent, formula [[2]],  time]  <> ")",
      Head[formula] === Until,  varsuffix  = RandomInteger[10ˆ5];
      "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣j" <>
       ToString[varsuffix ]  <> "␣>␣" <> ToString[time] <> "␣AND␣At(\""␣<>
␣␣␣␣␣␣␣ToString[agent]␣<>␣"\",␣j" <> ToString[varsuffix] <>
       ")␣AND␣(" <>
       auxlocal[agent, formula [[2]],  "j"  <> ToString[varsuffix]] <>
       ")␣AND␣FORALL(k" <> ToString[varsuffix] <> ":␣INT):␣((" <>
       ToString[time] <> "␣<␣k" <> ToString[varsuffix] <> ")␣AND␣(k" <>
        ToString[varsuffix ]  <> "␣<␣j" <> ToString[varsuffix] <>
       "))␣=>␣(At(\""␣<>␣ToString[agent]␣<>␣"\",␣k" <>
       ToString[varsuffix ]  <> ")␣=>␣(" <>
       auxlocal[agent, formula [[1]],  "k"  <> ToString[varsuffix]] <>
       "))",
      Head[formula] === Since, varsuffix = RandomInteger[10ˆ5];
      "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣j" <>
       ToString[varsuffix ]  <> ">=0␣AND␣j" <> ToString[varsuffix] <>
       "␣<␣" <> ToString[time] <> "␣AND␣j" <> ToString[varsuffix] <>
       "␣>=␣0␣AND␣At(\""␣<>␣ToString[agent]␣<>␣"\",␣j" <>
       ToString[varsuffix ]  <> ")␣AND␣(" <>
       auxlocal[agent, formula [[2]],  "j"  <> ToString[varsuffix]] <>
       ")␣AND␣FORALL(k" <> ToString[varsuffix] <> ":␣INT):␣((" <>
      ToString[time] <> "␣>␣k" <> ToString[varsuffix] <> ")␣AND␣(k" <>
       ToString[varsuffix] <> "␣>␣j" <> ToString[varsuffix] <>
      "))␣=>␣(At(\""␣<>␣ToString[agent]␣<>␣"\",␣k" <>
      ToString[varsuffix] <> ")␣=>␣(" <>
      auxlocal[agent, formula [[1]], "k" <> ToString[varsuffix]] <>
      "))",
    Head[formula] === Communication,
    "At(\""␣<>␣ToString[formula[[1]]]␣<>␣"\",␣" <> ToString[time] <>
      ")␣AND␣(" <> auxlocal[formula[[1]], formula [[2]], time] <> ")",
    Head[formula] === X, varsuffix = RandomInteger[10ˆ5];
    "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣j" <>
      ToString[varsuffix] <> "␣>␣" <> ToString[time] <> "␣AND␣At(\""␣<>
␣␣␣␣␣␣␣ToString[agent]␣<>␣"\",␣j" <> ToString[varsuffix] <>
      ")␣AND␣(" <>
      auxlocal[agent, formula [[1]], "j" <> ToString[varsuffix]] <>
      ")␣AND␣FORALL(k" <> ToString[varsuffix] <> ":␣INT):␣((" <>
      ToString[time] <> "␣<␣k" <> ToString[varsuffix] <> ")␣AND␣(k" <>
       ToString[varsuffix] <> "␣<␣j" <> ToString[varsuffix] <>
      "))␣=>␣(NOT␣At(\""␣<>␣ToString[agent]␣<>␣"\",␣k" <>
      ToString[varsuffix] <> "))",

    Head[formula] === F, varsuffix = RandomInteger[10ˆ5];
    "EXISTS(j" <> ToString[varsuffix] <> ":␣INT):␣j" <>
      ToString[varsuffix] <> "␣>␣" <> ToString[time] <> "␣AND␣(" <>
      auxlocal[agent, formula [[1]], "j" <> ToString[varsuffix]] <>
      ")␣AND␣At(\""␣<>␣ToString[agent]␣<>␣"\",␣j" <>
      ToString[varsuffix] <> ")",
    Head[formula] === G, varsuffix = RandomInteger[10ˆ5];
    "FORALL(i" <> ToString[varsuffix] <> ":␣INT):␣(i" <>
      ToString[varsuffix] <> "␣>␣" <> ToString[time] <> "␣AND␣At(\""␣<>
␣␣␣␣␣␣␣ToString[agent]␣<>␣"\",␣i" <> ToString[varsuffix] <>
      "))␣=>␣(" <>
      auxlocal[agent, formula [[1]], "i" <> ToString[varsuffix]] <> ")"

      ]]];
```