UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Universitat Politècnica de València
Dep. de Sistemas Informáticos y Computación

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Master Thesis

Logical Models for Automated Semantics-Directed Program Analysis

Candidate:
Patricio Reinoso Mendoza

Supervisors:
Salvador Lucas, Raúl Gutiérrez

July 2015

Partially supported by the EU (FEDER), Spanish MINECO project TIN 2013-45732-C4-1-P and GV project PROMETEO/2015/013.

Author's address:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

To my beloved wife Lucia, my soulmate and partner. This year studying abroad
becomes one of the greatest experiences in our lives, and I am thankful to have you
by my side supporting me in this endevour.
I love you.

Acknowledgments

I want to thank to Salvador Lucas and Raúl Gutiérrez. Their great experience and expertise in this area of knowledge, was invaluable during the project execution.

July, 2015

Patricio Reinoso Mendoza

Abstract

In Computer Science and Software Engineering, even at this time, when huge hardware and software resources are available, the problem of checking correctness of an specific piece of software is a very complicated one. Since the manual inspection of software is a difficult and error prone task, we propose as the main objective of this thesis the development of a tool which is able to generate logical models which can be used as a basis for semantics directed program analysis. To develop this tool, we rely on Order-Sorted First-Order Logic, which is the logic we use to define our programs and properties to be analyzed. We use this logic because it is sufficiently expressive to be used in the semantic description of most programming languages. Also, we use Convex Domain Interpretations as a flexible and computationally suitable basis for our derived models. We will also use polynomial interpretations and we will deal with conditional polynomial constraints, which are amenable for automating tests of properties written in order-sorted first-order logic. We have developed an automatic test tool, named AGES, which applies the aforementioned theoretical framework to implement the automated generation of models using convex domains and conditional polynomial constraints. The tool accepts Order-Sorted First-Order theories written in MAUDE, transforms them into a set of polynomial constraints, and then solves those constraints using an external SMT solver tool. The outcome of the constraint solver is used to assemble a logical model for the initial theory, which often can be used later to test other properties (termination, correctness, etc). The tool is written in Haskell to mutually exploit synergies between the functionalities provided by AGES and the functionalities provided by the MU-TERM tool.

Contents

List of Figures

1 Introduction

"Even when coping with many rules of good design and programming, programs may still contain errors" (Doron A. Peled)

In Computer Science, even at this time, with modern computer systems involving complex hardware and software, ensuring the correctness of the running software is a problem. This problem highly depends on the context of the application. For instance, in critical systems (e.g., a flight controller module inside an airplane, or a nuclear power plant control software), we need to guarantee total safety to all users of the service.

The manual inspection of complex software is usually a very expensive, time consuming, and error prone task. Here is where the need of automatic verification tools arise. Some tools try to find errors in the execution using *test sets* to detect conditions of the ejecution leading a given piece of code to fail. Formal verification tools check the program behaviour for every possible input condition. Such kind of tools, however, are in an early stage of maturity. In this thesis, we try to contribute to the development of such tools.

1.1 Automatic program analysis and verification

Manual software analysis and verification is a time consuming and error prone task. Analizing a small piece of software *by hand* is easy in some cases, but for complex software with thousand or maybe millions of lines of code, this is unfeasible. Furthermore, considering that some *mission critical* software requires total reliability, such a verification cannot be done by humans, and requires automated and reliable tools. Minimal requirements for such automatic verification tools are the following:

- The verification needs to be *correct*. This means that the conclusions drawn by the tool must fit the real behavior of the system, according to its formal definition.
- The verification process must be highly *automated* and *scalable* to handle complex software.

The first requirement, is the most important one: reliability of the analysis requires that the algorithms and theories used in the software verification must be correct. We can achieve this requirement by using *semantics-based* analysis techniques [16] so that correctness of the analysis is guaranteed by construction. The second requirement is more difficult to achieve. Even when full automation is possible, the time invested in the process cannot be excessive. Also, if the verification tool cannot work with *real* code, it will be useless in an industrial application.

Nowadays, automated verification tools are in their *childhood* stage. However, new theories and algorithms are developed in research laboratories and new tools are developed and implemented. The quest for tools that can be helpful for developers to create safe and correct industrial software continues.

1.2 Logical-based programming and program analysis

We are interested in developing general techniques for automated program analysis. For this purpose, some issues need to be clarified:

- What kind of *programs* are we interested in?
- What does *running* a program mean?
- Which *program properties* are at stake?
- How to *check* such properties in finite time?
- Is automation possible?

Logic is a unifying framework that provides answers to all these questions:

- A *program* is a *theory* $\mathcal{S}$ of a given logic $\mathcal{L}$ [15].
- *Running* a program is proving a specific kind of *sentence* (a *goal*) with respect to the theory $\mathcal{S}$ using the *inference system* of the logic $\mathcal{L}$ [15].
- We can use a logic language to encode *program properties* as sentences φ of a logic, possibly referring the theory $\mathcal{S}$ [14].
- *Checking* a program property φ can be seen as a *decision problem* for the logic $\mathcal{L}$ and theory $\mathcal{S}$.
- *Automation* is possible by using *decidable theories* where algorithms are available to check satisfaction of a formula [17].

As suggested by Goguen and Meseguer [6], *order-sorted first-order logic* (OS-FOL) provides a sufficiently powerful working framework for *declarative programming languages*. Essential ingredients inthe language of this logic are:

$$\text{(Rf)} \quad \frac{}{t \to^* t} \qquad\qquad \text{(T)} \quad \frac{t \to t' \qquad t' \to^* u}{t \to^* u}$$

$$\text{(C)} \quad \frac{t_i \to t'_i}{f(t_1, \ldots, t_i, \ldots, t_k) \to f(t_1, \ldots, t'_i, \ldots, t_k)} \quad \text{where } f \in \Sigma_{w,s},\ w = s_1, \ldots, s_k, \text{ and } 1 \leq i \leq k \qquad\qquad \text{(Re)} \quad \frac{}{\ell \to r} \quad \text{where } \ell \to r \in \mathcal{R}$$

Figure 1.1: Inference rules for Order-Sorted TRSs $\mathcal{R}$

- A set S of *sorts* which are ordered in a subsort relation $\leqslant$ which means *subset inclusion.*
- Sets $\Sigma_{w,s}$ and $\Pi_{s,s}$ of function and predicate symbols where $s \in S$ and w is a sequence $s_1 \ldots s_k$ of sorts from S.

We will use OS-FOL to represent declarative programs, semantics of programming languajes and properties.

Remark 1 An important special case of OS-FOL is the logic of *Order-Sorted Term Rewriting Systems* (OS-TRSs) where only two predicate symbols are allowed: $\to$ and $\to^*$. Such predicate symbols are used to describe computations with OS-TRSs as defined by means of the inference system in Figure 1.1. The programs considered in the examples given in this thesis are OS-TRSs which consist of sets $\mathcal{R}$ of rules $\ell \to r$, where ℓ and r are *terms* (more precise definitions are given below). ■

The algebraic specification and programming language Maude [2] supports OS-FOL expressivity. Of course, this does not mean that the use of OS-FOL is limited to such kind of languages. In the following, we focus on the development of techniques for automated analysis of programs in OS-FOL. First, we illustrate the approach with a simple example [10].

1.2.1 A running example: browsing a web site

As an example of program which can be seen as an order-sorted first-order *theory*, we will use the Maude program in Figure 1.2. This program represents the connectivity of a subset of web pages in the web site of the *First International Workshop on Automated Specification and Verification of Websites (WWV05).*

`http://www.dsic.upv.es/workshops/wwv05/`

The web pages are modeled as terms $p(u)$ where p is the specific web page and u represents the type of user who browses the web site, and the transitions between the pages are defined as *rewrite rules.* A screenshot of the homepage of the website can be found in Figure 1.3.

Web pages are classified in two types: regular pages, and restricted access pages. These are defined by terms of sort `WebPage` and `SecureWebPage`, respectively. The

```
mod WWV05-WEBSITE is
  sorts EventualUser RegUser User WebPage SecureWebPage .
  subsorts RegUser EventualUser < User .
  subsorts SecureWebPage < WebPage .

  ops login register sbmlink submission wwv05 : User -> WebPage .
  op vlogin : User -> SecureWebPage .
  op submit : RegUser -> SecureWebPage .

  var R : RegUser .
  var U : User .

  rl wwv05(U) => submission(U) .
  rl submission(U) => sbmlink(U) .
  rl sbmlink(U) => login(U) .
  rl sbmlink(U) => register(U) .
  rl login(U) => vlogin(U) .
  rl vlogin(U) => submit(R) .
  rl login(U) => vlogin(U) .
  rl vlogin(R) => submit(R) .
endm
```

Figure 1.2: Maude Specification of part of the WWV05 website

fact that secure web pages are special web pages is modeled by means of the subsort relation between them.

For the users, the situation is similar, because there are two subtypes of users: registered users and eventual users. They correspond to terms of sort `User`, `EventualUser` and `RegUser`, with their respective subsort relationship.

As an example of program property to be checked in this example, we consider the following:

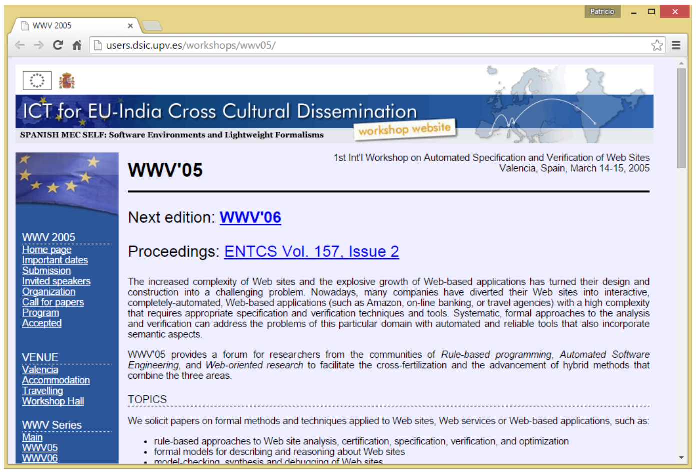
> *if u is able to reach the submission page from the main page, then u must be a registered user, and cannot be an eventual user.*

In other words, we want to guarantee secure access to special web pages for specific (registered) users only. We can express this property using OS-FOL sentences as well:

$$\forall \mathtt{u} : \mathtt{User}, \mathtt{wwv05(u)} \to^{*} \mathtt{submit(u)} \quad \Rightarrow \quad \neg(\mathtt{u} : \mathtt{EventualUser}) \qquad (1.1)$$

$$\forall \mathtt{u} : \mathtt{User}, \mathtt{wwv05(u)} \to^{*} \mathtt{submit(u)} \quad \Rightarrow \quad \mathtt{u} : \mathtt{RegUser} \qquad (1.2)$$

How do we *check* that these sentences hold? The following section discusses this problem according to [10].


Patricio
WWV 2005
users.dsic.upv.es/workshops/wwv05/
ICT for EU-India Cross Cultural Dissemination
workshop website

SPANISH MEC SELF: Software Environments and Lightweight Formalisms

WWV'05

1st Int'l Workshop on Automated Specification and Verification of Web Sites
Valencia, Spain, March 14-15, 2005

Next edition: WWV'06

Proceedings: ENTCS Vol. 157, Issue 2

The increased complexity of Web sites and the explosive growth of Web-based applications has turned their design and construction into a challenging problem. Nowadays, many companies have diverted their Web sites into interactive, completely-automated, Web-based applications (such as Amazon, on-line banking, or travel agencies) with a high complexity that requires appropriate specification and verification techniques and tools. Systematic, formal approaches to the analysis and verification can address the problems of this particular domain with automated and reliable tools that also incorporate semantic aspects.

WWV'05 provides a forum for researchers from the communities of *Rule-based programming*, *Automated Software Engineering*, and *Web-oriented research* to facilitate the cross-fertilization and the advancement of hybrid methods that combine the three areas.

TOPICS

We solicit papers on formal methods and techniques applied to Web sites, Web services or Web-based applications, such as:

- rule-based approaches to Web site analysis, certification, specification, verification, and optimization
- formal models for describing and reasoning about Web sites
- model-checking, synthesis and debugging of Web sites

WWV 2005
Home page
Important dates
Submission
Invited speakers
Organization
Call for papers
Program
Accepted

VENUE
Valencia
Accommodation
Travelling
Workshop Hall

WWV Series
Main
WWV05
WWV06

Figure 1.3: WWV05 Website Home Page

1.3 Models for Order-Sorted First-Order Logic

The automatic generation of *models* of theories (in this case, for Order-Sorted First-Order Logic) is essential in automated analysis. Models are just algebraic structures $\mathcal{A}$ interpreting the syntactic components of the language of a logic $\mathcal{L}$ [9]: sorts s are interpreted as sets of values $\mathcal{A}_s$, usually called *domains*; function symbols f as *mappings* $f_\mathcal{A}$ taking and returning values in the corresponding domains; predicate symbols P are interpreted as *relations* among the semantic domains. Given a theory $\mathcal{S}$ of a logic $\mathcal{L}$, we say that a structure $\mathcal{A}$ is a model of $\mathcal{S}$ if all formulae in $\mathcal{S}$ are true when interpreted according to $\mathcal{A}$ (precise definitions are given below).

Models are important because they provide a way to *simulate*, with some degree of *abstraction*, the computational relations (that can be the one step transition $\to$, many steps transition $\to^*$, etc.) associated to programs, and defined by means of the inference system of a computational logic. Also, we can use these automatically generated components of the models (relations, interpretations of function symbols, etc.) to implement proofs in semantics-based and semantics-directed program analysis.

Remark 2 Semantics-based program analysis are those where *the information provided by the analysis can be proved to be safe (or correct) with respect to a semantics of the programming language* [16, page 3]. We call it *semantics-directed*, if it is not only based on the semantics of the programming language but also *the structure of the analysis reflects the structure of the semantics* (cf. [16, page 3]). ■

In program analysis, *semantic structures* $\mathcal{A}'$ [9] leading to *decidable theories* $Th(\mathcal{A}')$ [17] can be used to provide an effective way to *find* logical models $\mathcal{A}$ for the specification $\mathcal{S}$ and also to *check* whether the properties at stake (regarding $\mathcal{S}$) hold. This is often possible by using simple *theory transformations* κ from the language of $\mathcal{S}$

into the language of $Th(\mathcal{A}')$ to obtain a set of sentences $\mathcal{S}' = \kappa(\mathcal{S})$ which is then *decidable*. This has been formalized in [10] by extending the notion of *derived algebra* [8] to logical structures.

Example 3

The interpretation of sorts `User`, `RegUser` and `EventualUser` as $[0, 1]$, $\{0\}$, and $\{1\}$, respectively, the *sort inclusion* expressions in the specification `WWV05-WEBSITE` as *set memberships*, together with the interpretation of symbols as *arithmetic operations* (for instance, `wwv05` interpreted as the identity function and `submit` as $x + x$), and the interpretation of $\to^*$ (viewed as a binary predicate) as the *equality* of natural numberes, lead to the translation of formulae (3.19) and (3.20) into the *arithmetic* sentences

$$\forall u \in [0, 1], u = u + u \quad \Rightarrow \quad u \neq 1$$

and

$$\forall u \in [0, 1], u = u + u \quad \Rightarrow \quad u = 0$$

respectively. It is not difficult to see that, under the *intended interpretation* of the arithmetic symbols in there, both are actually equivalent to

$$\forall u \in [0, 1], u = u + u \quad \Rightarrow \quad u = 0 \tag{1.3}$$

which is *true*. This is an example of how a *derived model* for a theory `WWV05-WEBSITE` can be obtained from an *intended model* of a transformed theory (essentially arithmetic, in this case). The complete technical details are developed in [10] and further discussed below with regard to the *mechanization* of the whole proof.

1.4 Automatic generation of models

The *human-guided* Example 3 follows a *top-down* approach that starts with an human-defined structure $\mathcal{A}$, which is used to check that it is a model of $\mathcal{S}$. The *automatic generation* of a model is rather a *bottom-up* process, where everything remains unspecified until we can obtain a succesful attemp to solve constraints obtained from $\mathcal{S}$. The obtained solution is used later to synthetize a structure which represents a model of $\mathcal{S}$. This is acomplished as follows:

Parameterization

The components of the model are made *parametric* so that from a single collection of similar sets, functions, etc., we can obtain different instancies by giving specific values to the parameters.

1. Sorts $s \in S$ are interpreted as specific cases $\mathcal{A}_s$ of a class of domains. In particular, we use the *convex domains* introduced in [11] and the corresponding

adaptation to the order-sorted setting discussed in [10]. Convex domains $D(\mathsf{C},\vec{b})$ are sets of vectors $\vec{x} \in \mathbb{R}^n$ which are defined by means inequalities $\mathsf{C}\vec{x} \geq \vec{b}$ for some matrix C and vector $\vec{b}$, i.e., $D(\mathsf{C},\vec{b}) = \{x \in \mathbb{R}^n \mid \mathsf{C}\vec{x} \geq \vec{b}\}$. The advantage of convex domains is that are defined by means of linear inequalitites which are easy to handle by means of standard techniques from linear algebra [18]. Examples of convex domains are

- the emptyset $D(\mathsf{C},\vec{b}) = \emptyset$ if $\mathsf{C} = (0)$ and $\vec{b} = (1)$;
- the singleton $\{0\}$: $D(\mathsf{C},\vec{b}) = \{0\}$ if $\mathsf{C} = (1,-1)^T$ and $\vec{b} = (0,0)^T$;
- the interval $[0,1]$: $D(\mathsf{C},\vec{b}) = [0,1]$ if $\mathsf{C} = (1,-1)^T$ and $\vec{b} = (0,-1)^T$; and
- the set of non-negative numbers $[0,+\infty)$: $D(\mathsf{C},\vec{b}) = [0,+\infty)$ if $\mathsf{C} = (1)$ and $\vec{b} = (0)$.

All these domains are particular cases of two kinds of parametric convex domains: $D(C_1,\vec{b_1})$ where $C_1 = (c_1)$ and $\vec{b}_1 = (b_1)$ for parameters c_1 and b_1 ranging on appropriate (usually finite or bounded, for automation purposes) subsets of real numbers, and $D(C_2,\vec{b_2})$ where $C_2 = (c'_1, c'_2)^T$ and $\vec{b}_2 = (b'_1, b'_2)$, where c'_1, c'_2, b'_1 and b'_2 are also parameters.

2. The syntactic objects are given *parametric interpretations* of a given type, usually chosen according to their amenability to automation. For instance, function symbols can be given *linear polynomials* $a_1x_1 + a_2x_2 + \cdots + a_kx_k + a_0$, where $a_0, a_1, \ldots, a_k$ are *parameters*. More general polynomials, or other kind of functions can be considered.

3. *Sentences* $\phi \in \mathcal{S}$ are used to obtain a new set $\mathcal{S}'$ of *parametric* sentences $\phi^\sharp$ with existentially quantified parameters $a_1, \ldots, a_n$. Such parameters range over appropriate domains $D_1, \ldots, D_n$.

4. $\mathcal{S}'$ is treated as a *constraint* whose solutions $\sigma = \{a_i \mapsto d_i \mid 1 \leq i \leq n\}$ make $\sigma(\phi^\sharp)$ (an instantiation of the parameters in $\phi^\sharp$) *true*.

Example 4

Although sentence (1.3) in Example 3 (call it ϕ) clearly holds, it is usually desirable to introduce some *flexibility* in the definition of the derived model so that we succeed more often. Typically, this is achieved by introducing *parameters* whose value can be freely given, often by means of some controlled *search process* (for automation). For instance, instead of ϕ, we could write the following *parametric sentence* $\phi^\sharp$

$$\forall u \in [0,1], u = a_1u + a_2u \quad \Rightarrow \quad u = 0 \tag{1.4}$$

where a_1 and a_2 range over the naturals (for instance). Clearly, ϕ is a *particular case* of $\phi^\sharp$, if $a_1 = a_2 = 1$. Note, however, that $a_1 = a_2 = 0$ does *not* make it true!

In general, the satisfaction of a derived sentence may *depend* on the specific interpretation which is used for the symbols. Parameters are useful to bring flexibility into the definition of a model so that we can ‘recover’ from a failing attempt to find a model: just try another combination of parameters. Typically, this is *implemented* by means of some *constraint solving* mechanism that seeks for appropriate *instantiations* σ of the parameters of a parametric formula $\phi^\sharp$ in a suitable, possibly finite, set of values in such a way that the instantiated formula $\sigma(\phi^\sharp)$ true.

In the following section, we end this sketch of mechanization techniques by briefly explaining how to deal with *implications* (like the ones occurring in the previous examples) in a systematic way which, together with constraint solving mechanisms, is essential for a full automation of model generation.

Dealing with implications

An important issue is handling formulae containing *implications* as in the previous examples. For sentences involving arithmetical expressions only, a number of techniques from linear algebra and real algebraic geometry are helpful for that. In particular the Affine form of Farkas’ Lemma considered in [11, Section 5.1], can be used to deal with linear conditional constraints like those in Examples 3 and 4 that consist of *implications* $\bigwedge_{j=1}^{p_i} e_{ij} \geq d_{ij} \Rightarrow e_i \geq d_i$, where for all $i \in \{1,\ldots,k\}$, $p_i > 0$ and for all j, $1 \leq j \leq p_i$, e_{ij} and e_i are linear expressions[1] and $d_{ij}, d_i \in \mathbb{R}$. We say that these implications are in *affine form*. In general, given $\vec{c} \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$, the affine form of Farkas’ Lemma can be used to check whether a constraint $\vec{c}^T\vec{x} \geq \beta$ holds whenever $\vec{x}$ ranges on the set S of solutions $\vec{x} \in \mathbb{R}^n$ of a linear system $A\vec{x} \geq \vec{b}$ of k inequalities, i.e., A is a matrix of k rows and n columns and $\vec{b} \in \mathbb{R}^k$. According to Farkas’ Lemma, we have to *find* a vector $\vec{\lambda}$ of k non-negative numbers $\vec{\lambda} \in \mathbb{R}^k_0$ such that $\vec{c} = A^T\vec{\lambda}$ and $\vec{\lambda}^T\vec{b} \geq \beta$.

Example 5

We can write (1.4) to fit the *affine* form above as follows:

$$u \geq 0 \wedge -u \geq -1 \wedge (1-a_1-a_2)u \geq 0 \wedge (a_1+a_2-1)u \geq 0 \quad \Rightarrow \quad u \geq 0 \qquad (1.5)$$
$$u \geq 0 \wedge -u \geq -1 \wedge (1-a_1-a_2)u \geq 0 \wedge (a_1+a_2-1)u \geq 0 \quad \Rightarrow \quad -u \geq 0 \qquad (1.6)$$

Note that only u is considered a *variable* in the linear expressions; a_1 and a_2 are viewed as *parameters* and $1-a_1-a_2$ and a_1+a_2-1 are viewed as *coefficients* accompanying variable u in the linear expression. Note also that we obtain a logically equivalent set of two implications due to the conjunction of two affine inequalities $u \geq 0 \wedge -u \geq 0$ which are equivalent to $u = 0$ in the consequent of (1.4). Now, we apply Farkas’ lemma to each of them. Both (1.5) and (1.6) have the same associated matrix A, which is actually a four components *vector* (a row per inequality in the antecedent of the implications (1.5) and (1.6))

$$(1, -1, 1-a_1-a_2, a_1+a_2-1)^T.$$

[1]note that $e = e'$ if and only if $e \geq e'$ and $e' \geq e$

Similarly, we have the same vector $\vec{b} = (0, -1, 0, 0)^T$ in both cases. For (1.5), $\vec{c}$ is actually a one-dimensional vector (1) and $\beta = 0$. For (1.6) $\vec{c}$ is (-1) and $\beta = 0$.

For (1.5), the application of Farkas' Lemma seeks a vector $\vec{\lambda} = (\lambda_1, \lambda_2, \lambda_3, \lambda_4)^T$ with $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0$ that satisfies the following two (in)equations:

$$1 = \lambda_1 - \lambda_2 + (1 - a_1 - a_2)\lambda_3 + (a_1 + a_2 - 1)\lambda_4 \qquad -\lambda_2 \geq 0$$

for some values of the parameters a_1 and a_2. Note that, since $\lambda_2 \geq 0$, the second inequality imposes $\lambda_2 = 0$. Hence, we have to solve the equality

$$1 = \lambda_1 + (1 - a_1 - a_2)\lambda_3 + (a_1 + a_2 - 1)\lambda_4$$

For (1.6), we seek some $\vec{\lambda}' = (\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4)^T$ with $\lambda'_1, \lambda'_2, \lambda'_3, \lambda'_4 \geq 0$ that satisfies

$$-1 = \lambda'_1 - \lambda'_2 + (1 - a_1 - a_2)\lambda'_3 + (a_1 + a_2 - 1)\lambda'_4 \qquad -\lambda'_2 \geq 0$$

or, as before,

$$-1 = \lambda'_1 + (1 - a_1 - a_2)\lambda'_3 + (a_1 + a_2 - 1)\lambda'_4$$

for some values of the parameters a_1 and a_2. Note, however, that the values associated to a_1 and a_2 should be *the same* in both cases as they represent ingredients defining *the same* semantic structure. Thus, we need a *single* solution for the set of equations

$$\begin{array}{rcl} 1 & = & \lambda_1 + (1 - a_1 - a_2)\lambda_3 + (a_1 + a_2 - 1)\lambda_4 \\ -1 & = & \lambda'_1 + (1 - a_1 - a_2)\lambda'_3 + (a_1 + a_2 - 1)\lambda'_4 \end{array}$$

A solution is

$$a_1 = a_2 = 1 \qquad \lambda_1 = \lambda_3 = \lambda'_3 = \lambda_4 = 1 \qquad \lambda'_1 = \lambda'_4 = 0.$$

Another solution is

$$a_2 = 2 \qquad a_1 = 0 \qquad \lambda_1 = \lambda_3 = 2 \qquad \lambda'_3 = \lambda_4 = 1 \qquad \lambda'_1 = \lambda'_4 = 0$$

These solutions can be obtained by using automatic tools for arithmetic constraint solving like MultiSolver[2], as we used in this example.

1.5 Plan of the thesis

This thesis develops a first implementation of the logical-based verification approach sketched above for order-sorted first-order logic. We follow the formalization developed in [10, 11], which supports the methodology described above. This thesis, though, also provides an account of several practical issues concerning the *mechanization* of the approach which are useful to clarify and better understand how to use it.

The main outcome of the thesis is the tool AGES, which stands for

[2] http://zenon.dsic.upv.es/multisolver/

Automatic GEneration of logical modelS

and generates logical models for order-sorted first-order theories. The tool generates models using the convex domains technique. The tool is useful to validate handcrafted solutions of considered problems, and also to extend the analysis in other directions.

This tool is able to:

- Read an Order-Sorted First-Order theory $\mathcal{S}$ given as a Maude program.
- Read a Goal to be tested.
- Parse the theory and the goal into a MU-TERM variable
- Generate a set of logic formulae from the input theory, the inference rules and the goal.
- Implements the generation of *convex domains* and the use of *convex matrix interpretations* in the generation of models.
- Generate arithmetic constraints and solve them using an SMT solver.
- Return the obtained model.

The techniques developed in the tool (in particular, the use of *convex domains* and *convex matrix interpretations*) have also been incorporated into the termination tool MU-TERM [1], which was created to verify termination properties of variations of *Term Rewriting System* (TRS). In [10, 11] examples of the use of convex domains to prove termination have been given.

This thesis is divided in five chapters. Chapter 1 includes an introduction to the automatic and logical-based software analysis. Chapter 2 contains the preliminaries. Chapter 3 contains the formal treatment of a verification problem. Chapter 4 and Chapter 5 explain the practical implementation of the tool, including all the steps to setting up the developer environment, the Haskell modules taken from MU-TERM to use in the new tool, the new modules created, and all the issues and problems that appear during the developing process. Chapter 6 closes this work with the conclusions, and future work.

2 Preliminaries

In this chapter, we introduce the basics of order-sorted first-order logic, as we need for the development of the thesis. We follow [6, 7]. We also introduce the main definitions and facts regarding *convex domains* and convex matrix interpretations, according to [10, 11].

2.1 Order-Sorted First-Order Logic

2.1.1 Sorts and order-sorted signatures

Given a set of *sorts* S, a many-sorted signature is an $S^* \times S$-indexed family of sets $\Sigma = \{\Sigma_{w,s}\}_{(w,s)\in S^*\times S}$ containing *function symbols* with a given string of argument sorts and a result sort. If $f \in \Sigma_{s_1\cdots s_n,s}$, then we display f as $f : s_1 \cdots s_n \to s$.

Constant symbols c (taking no argument) have rank declaration $c : \lambda \to s$ for some sort s (where λ denotes the *empty* sequence).

An order-sorted signature $(S, \leq, \Sigma)$ consists of a *partially ordered set* (poset) of sorts $(S, \leq)$ together with a many-sorted signature (S, Σ). The *connected components* of $(S, \leq)$ are the equivalence classes $[s]$ corresponding to the least equivalence relation $\equiv_\leq$ containing $\leq$.

We extend the order $\leq$ on S to strings of equal length in S^* by $s_1 \cdots s_n \leq s'_1 \cdots s'_n$ iff $s_i \leq s'_i$ for all i, $1 \leq i \leq n$.

Symbols f can be *subsort-overloaded*, i.e., they can have several rank declarations related in the $\leq$ ordering [7]. Constant symbols, however, have only one rank declaration. Besides, the following *monotonicity condition* must be satisfied: $f \in \Sigma_{w_1,s_1} \cap \Sigma_{w_2,s_2}$ and $w_1 \leq w_2$ imply $s_1 \leq s_2$.

To avoid ambiguous terms, we assume that Σ is *sensible*, meaning that if $f : s_1 \cdots s_n \to s$ and $f : s'_1 \cdots s'_n \to s'$ are such that $[s_i] = [s'_i]$, $1 \leq i \leq n$, then $[s] = [s']$.

An order-sorted signature Σ is *regular* iff given $w_0 \leq w_1$ in S^* and $f \in \Sigma_{w1,s_1}$, there is a least $(w, s) \in S^* \times S$ such that $f \in \Sigma_{w,s}$ and $w_0 \leq w$.

If, in addition, each connected component of the sort poset has a top element, then the regular signature is called *coherent*.

Example 6

The order-sorted signature $(S, \leq, \Sigma)$ for `WWV05-WEBSITE` in Figure 1.2 consists of the following components:

- *Set of sorts* $S = \{\texttt{EventualUser}, \texttt{RegUser}, \texttt{User}, \texttt{SecureWebPage}, \texttt{WebPage}\}$.
- The *subsort relation* is the least ordering $\leq$ on S satisfying

  $\texttt{EventualUser} \leq \texttt{User}$; $\texttt{RegUser} \leq \texttt{User}$; and $\texttt{SecureWebPage} \leq \texttt{WebPage}$.

- $\Sigma = \Sigma_{\texttt{User},\texttt{WebPage}} \cup \Sigma_{\texttt{User},\texttt{SecureWebPage}} \cup \Sigma_{\texttt{RegUser},\texttt{SecureWebPage}}$, with

$$\begin{array}{rcl} \Sigma_{\texttt{User},\texttt{WebPage}} & = & \{\texttt{login}, \texttt{register}, \texttt{sbmlink}, \texttt{submission}, \texttt{wwv05}\} \\ \Sigma_{\texttt{User},\texttt{SecureWebPage}} & = & \{\texttt{vlogin}\} \\ \Sigma_{\texttt{RegUser},\texttt{SecureWebPage}} & = & \{\texttt{submit}\} \end{array}$$

  Note that there is no constant symbol.

The set of variables is $\mathcal{X} = \mathcal{X}_{\texttt{RegUser}} \cup \mathcal{X}_{\texttt{User}}$, with $\mathcal{X}_{\texttt{RegUser}} = \{\texttt{R}\}$, and $\mathcal{X}_{\texttt{User}} = \{\texttt{U}\}$.

An order-sorted signature *with predicates* is a quadruple $(S, \leq, \Sigma, \Pi)$ such that $(S, \leq, \Sigma)$ is an coherent order-sorted signature, and $\Pi = \{\Pi_w \mid w \in S^+\}$ is a family of *predicate symbols* P, Q, ...

We can use Σ, Π instead of $(S, \leq, \Sigma, \Pi)$ if S and $\leq$ are clear from the context.

2.1.2 Theories, Specifications and programs.

A *theory* $\mathcal{S}$ of Σ, Π is a set of formulae, $\mathcal{S} \subseteq \mathit{Form}_{\Sigma,\Pi}$, and its *theorems* are the formulae $\varphi \in \mathit{Form}_{\Sigma,\Pi}$ for which we can derive a proof using an appropriate inference system $\mathcal{I}(\mathcal{L})$ of a logic $\mathcal{L}$ in the usual way (written $\mathcal{S} \vdash \varphi$).

Given a logic $\mathcal{L}$ describing computations in a (declarative) programming language, programs are viewed as *theories* $\mathcal{S}$ of $\mathcal{L}$.

2.1.3 Structures, Satisfaction, Models

Given an order-sorted signature with predicates $(S, \leq, \Sigma, \Pi)$, an $(S, \leq, \Sigma, \Pi)$-*structure*[1] (or just a Σ, Π-structure) is an order-sorted $(S, \leq, \Sigma)$-algebra $\mathcal{A}$ together with an assignment to each $P \in \Pi_w$ of a subset $P_{\mathcal{A}} \subseteq \mathcal{A}_w$ such that [6]: (i) for P the identity predicate $\_ = \_ : ss$, the assignment is the identity relation, i.e., $(=)_{\mathcal{A}} = \{(a, a) \mid a \in \mathcal{A}_s\}$; and (ii) whenever $P : w_1$ and $P : w_2$ and $w_1 \leq w_2$, then $P_{\mathcal{A}_{w_1}} = \mathcal{A}_{w1} \cap P_{\mathcal{A}_{w_2}}$.

Let $(S, \leq, \Sigma, \Pi)$ be an order-sorted signature with predicates and $\mathcal{A}, \mathcal{A}'$ be $(S, \leq, \Sigma, \Pi)$-structures.

Then, an $(S, \leq, \Sigma, \Pi)$-*homomorphism* $h : \mathcal{A} \to \mathcal{A}'$ is an $(S, \leq, \Sigma)$-homomorphism such that, for each $P : w$ in Π, if $(a_1, \ldots, a_n) \in P_{\mathcal{A}}$, then $h(a_1, \ldots, a_n) \in P_{\mathcal{A}'}$.

[1]As in [9], we use 'structure' and reserve the word 'model' to refer those structures satisfying a given theory.

Given an S-sorted *valuation mapping* $\alpha : \mathcal{X} \to \mathcal{A}$, the evaluation mapping $[\_]^{\alpha}_{\mathcal{A}} : \mathcal{T}_{\Sigma}(\mathcal{X}) \to \mathcal{A}$ is the unique $(S, \leq, \Sigma)$-homomorphism extending α [7].

Finally, $[\_]^{\alpha}_{\mathcal{A}} : \mathit{Form}_{\Sigma,\Pi} \to \mathit{Bool}$ is given by:

1. $[P(t_1, \ldots, t_k)]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ if and only if $([t_1]^{\alpha}_{\mathcal{A}}, \ldots, [t_k]^{\alpha}_{\mathcal{A}}) \in P_{\mathcal{A}}$;
2. $[\neg\phi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ if and only if $[\phi]^{\alpha}_{\mathcal{A}} = \mathsf{false}$;
3. $[\phi \wedge \psi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ if and only if $[\phi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ and $[\psi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$;
4. $[(\forall x : s)\ \phi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ if and only if for all $a \in \mathcal{A}_s$, $[\phi]^{\alpha[x \mapsto a]}_{\mathcal{A}} = \mathsf{true}$;

We say that $\mathcal{A}$ *satisfies* $\varphi \in \mathit{Form}_{\Sigma,\Pi}$ if there is $\alpha \in \mathcal{X} \to \mathcal{A}$ such that $[\varphi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$. If $[\varphi]^{\alpha}_{\mathcal{A}} = \mathsf{true}$ for *all* valuations α, we write $\mathcal{A} \models \varphi$ and say that $\mathcal{A}$ is a *model* of φ [9, page 12].

Initial valuations are not relevant for establishing the satisfiability of *sentences*; thus, both notions coincide on them.

We say that $\mathcal{A}$ is *a model of a set of sentences* $\mathcal{S} \subseteq \mathit{Form}_{\Sigma,\Pi}$ (written $\mathcal{A} \models \mathcal{S}$) if for all $\varphi \in \mathcal{S}$, $\mathcal{A} \models \varphi$.

And, given a sentence φ, we write $\mathcal{S} \models \varphi$ if and only if for *all models* $\mathcal{A}$ of $\mathcal{S}$, $\mathcal{A} \models \varphi$.

Sound logics guarantee that every provable sentence φ is true in *every model* of $\mathcal{S}$, i.e., $\mathcal{S} \vdash \varphi$ implies $\mathcal{S} \models \varphi$.

2.1.4 Derived Models

Appropriate Σ-algebras can be obtained as *derived algebras* if we first consider a *new* signature Σ' of symbols with 'intended' (often arithmetic) interpretations.

Definition 7 (Derivor and Derived algebra) [8, Definition 11] *Let $\Sigma = (S, \leq, \Sigma)$ and $\Sigma' = (S', \leq', \Sigma')$ be order-sorted signatures. A* derivor *from Σ to Σ' is a monotone function $\tau : S \to S'$ (i.e., such that for all $s, s' \in S$, $s \leq s'$ implies*[2] *$\tau(s) \leq' \tau(s')$) and a family $d_{w,s} : \Sigma_{w,s} \to (\mathcal{T}_{\Sigma})_{\tau(w),\tau(s)}$, where $\tau(s_1, \ldots, s_k) = \tau(s_1), \ldots, \tau(s_k)$ and where $(\mathcal{T}_{\Sigma})_{\tau(w),\tau(s)}$ denotes the set of all Σ'-terms using variables $\{y_1, \ldots, y_k\}$ with y_i of sort $\tau(s_i)$. Each operation symbol $f \in \Sigma_{w,s}$ is expressed using a derived operation $d_{w,s}(f)$ of the appropriate arity. We often use d to denote a derivor $\langle \tau, d \rangle$. Now, let $\mathcal{A}'$ be an Σ'-algebra. Then, the d-derived algebra $d\mathcal{A}'$ of $\mathcal{A}'$ is the Σ-algebra with carriers $(d\mathcal{A})_s = \mathcal{A}'_{\tau(s)}$ for all $s \in S$; and mappings $f_{d\mathcal{A}'}$ for each $f \in \Sigma$ defined to be $(d(f))_{\mathcal{A}'}$, the derived operator of the Σ'-term $d(f)$.*

Example 8

Let $(S, \leq, \Sigma(\mathcal{X}))$ as in Example 6. Let $S' = \{\mathsf{zero}, \mathsf{one}, \mathsf{zeroOne}, \mathsf{nat}\}$ with subsort relation $\leq'$ given by

$$\mathsf{zero}, \mathsf{one} \leq' \mathsf{zeroOne} \leq' \mathsf{nat}$$

[2]Monotonicity is *not* required in [8] where only many-sorted signatures are considered.

and

$$\Sigma' = \Sigma'_{\lambda,\mathsf{zero}} \cup \Sigma'_{\lambda,\mathsf{one}} \cup \Sigma'_{\mathsf{nat}^2\mathsf{nat}}$$

where $\Sigma'_{\lambda,\mathsf{zero}} = \{\mathsf{0}\}$, $\Sigma'_{\lambda,\mathsf{one}} = \{\mathsf{1}\}$, and $\Sigma'_{\mathsf{nat}^2\mathsf{nat}} = \{+\}$ are symbols with the standard *intended meaning* as arithmetic constants and operations.

We define a *derivor* by

$$\begin{array}{rcl}
\tau(\mathtt{EventualUser}) & = & \mathsf{one} \\
\tau(\mathtt{RegUser}) & = & \mathsf{zero} \\
\tau(\mathtt{User}) & = & \mathsf{zeroOne} \\
\tau(\mathtt{WebPage}) & = & \mathsf{nat} \\
\tau(\mathsf{SecureWebPage}) & = & \mathsf{nat}
\end{array}$$

Also,

$$\begin{array}{rcll}
d(r) & = & \mathsf{0} & \text{for all } r \in \mathcal{X}_{\mathtt{RegUser}}, \\
d(x) & = & \mathsf{1} & \text{for all } x \in \mathcal{X}_{\mathtt{EventualUser}} \cup \mathcal{X}_{\mathtt{User}} \cup \mathcal{X}_{\mathtt{SecureWebPage}} \cup \mathcal{X}_{\mathtt{WebPage}} \\
d(\mathtt{submit}) & = & x + x & \\
d(f) & = & x & \text{for all other symbols } f \in \Sigma \text{ (all of them monadic!).}
\end{array}$$

Let $\mathcal{A}'$ be the $(S', \leq', \Sigma')$ algebra given by

$$\mathcal{A}'_{\mathsf{zero}} = \{0\},\ \mathcal{A}'_{\mathsf{one}} = \{1\},\ \mathcal{A}'_{\mathsf{zeroOne}} = \{0, 1\},\ \text{and } \mathcal{A}'_{\mathsf{nat}} = \mathbb{N}$$

together with the *standard* interpretations for 0, 1, and +. The derived $(S, \leq, \Sigma)$-algebra $\mathcal{A} = d\mathcal{A}'$ is given by

$$\mathcal{A}_{\mathtt{EventualUser}} = \mathcal{A}'_{\mathsf{one}} = \{1\},\ \mathcal{A}_{\mathtt{RegUser}} = \mathcal{A}'_{\mathsf{zero}} = \{0\},$$
$$\mathcal{A}_{\mathtt{User}} = \mathcal{A}'_{\mathsf{zeroOne}} = \{0, 1\},\ \text{and } \mathcal{A}_{\mathtt{SecureWebPage}} = \mathcal{A}_{\mathtt{WebPage}} = \mathcal{A}'_{\mathsf{nat}} = \mathbb{N},$$

together with the derived interpretations for each symbol in $\Sigma(\mathcal{X})$.

Generalizing Definition 7 we get the notion of *derived structure* [10, Definition 2].

Definition 9 (Derivor for signatures with predicates / Derived structure) *Let $\Sigma = (S, \leq, \Sigma, \Pi)$ and $\Sigma' = (S', \leq', \Sigma', \Pi')$ be order-sorted signatures with predicates and $\langle \tau, d \rangle$ be a derivor from $(S, \leq, \Sigma)$ to $(S', \leq', \Sigma')$. We extend d to predicate symbols by adding a component $d : \Pi \to \mathsf{Form}_{\Sigma'\Pi'}$ such that for all $P \in \Pi_w$, with $w = s_1 \cdots s_n$, $d(P)$ is an* atom *$P'(t'_1, \ldots, t'_m)$ with $P' \in \Pi'$, and terms $t'_1, \ldots, t'_m \in \mathcal{T}_{\Sigma'}(\mathcal{X})$ only use variables $\{y_1, \ldots, y_n\}$ with y_i of sort $\tau(s_i)$. In this new context we also call $\langle \tau, d \rangle$ a derivor. Let $\mathcal{A}' = (\mathcal{A}', \Sigma_{\mathcal{A}'}, \Pi'_{\mathcal{A}'})$ be an $(S', \leq', \Sigma', \Pi')$-structure and $\mathcal{A}'_0 = (\mathcal{A}', \Sigma_{\mathcal{A}'})$ be the underlying $(S', \leq', \Sigma')$-algebra. Then, the $\langle \tau, d \rangle$-derived structure $d\mathcal{A}'$ of $\mathcal{A}'$ is the $(S, \leq, \Sigma, \Pi)$-structure that consists of the Σ-algebra $d\mathcal{A}'_0$ with S-sorted set of carriers $\mathcal{A}$ together with interpretations $P_{d\mathcal{A}'}$ (for $P \in \Pi_w$) defined to be*

$$\begin{array}{rcl}
P_{d\mathcal{A}'} & = & \{([t_1]^{\alpha}_{d\mathcal{A}'}, \ldots, [t_n]^{\alpha}_{d\mathcal{A}'}) \mid (t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma}(\mathcal{X})_w, \alpha \in \mathcal{X} \to \mathcal{A}, \\
& & \quad d(P) = P'(t'_1, \ldots, t'_m), \mathcal{Y} = \mathcal{V}ar(t'_1, \ldots, t'_m), \sigma(y_i) = t_i, 1 \leq i \leq n \\
& & \quad \exists \alpha' : \mathcal{Y} \to \mathcal{A}'([\sigma(t'_1)]^{\alpha'}_{\mathcal{A}'}, \ldots, [\sigma(t'_m)]^{\alpha'}_{\mathcal{A}'}) \in P'_{\mathcal{A}'}\}
\end{array}$$

Note that $\langle \tau, d \rangle$ can be seen now as a transformation $d : \mathit{Form}_{\Sigma,\Pi} \rightarrow \mathit{Form}_{\Sigma',\Pi'}$:

$$\begin{array}{rcl} d(P(t_1,\ldots,t_n)) & = & d(P)[y_1 \mapsto d(t_1),\ldots,y_n \mapsto d(t_n)] \\ d(\neg\phi) & = & \neg d(\phi) \\ d(\phi \wedge \phi') & = & d(\phi) \wedge d(\phi') \\ d((\forall x : s)\phi) & = & (\forall x : \tau(s))d(\phi) \end{array}$$

The following results formalize the use of the previous construction.

Theorem 10 [10] *Let $\Sigma = (S, \leq, \Sigma, \Pi)$ and $\Sigma' = (S', \leq', \Sigma', \Pi')$ be order-sorted signatures with predicates and $\langle \tau, d \rangle$ be a derivor from $(S, \leq, \Sigma, \Pi)$ to $(S', \leq', \Sigma', \Pi')$. Let $\mathcal{A}'$ be an $(S', \leq', \Sigma', \Pi')$-structure and $\varphi \in \mathsf{Form}_{\Sigma,\Pi}$. If $\mathcal{A}' \models d(\varphi)$, then $d\mathcal{A}' \models \varphi$.*

Corollary 11 (Derived model) [10] *Let $\Sigma = (S, \leq, \Sigma, \Pi)$ and $\Sigma' = (S', \leq', \Sigma', \Pi')$ be order-sorted signatures with predicates and $\langle \tau, d \rangle$ be a derivor from $(S, \leq, \Sigma, \Pi)$ to $(S', \leq', \Sigma', \Pi')$. Let $\mathcal{A}'$ be an $(S', \leq', \Sigma', \Pi')$-structure and $\mathcal{S} \subseteq \mathit{Forms}_{\Sigma,\Pi}$ be a theory. If for all $\varphi \in \mathcal{S}$, $\mathcal{A}' \models d(\varphi)$, then $d\mathcal{A}' \models \mathcal{S}$.*

2.2 Convex Domains

We use the convex domains in [11] as the basis for our derived models. In the following, we recall the main concepts and definitions at stake.

2.2.1 Domains

Definition 12 *Given a matrix $\mathsf{C} \in \mathbb{R}^{m \times n}$, and $\vec{b} \in \mathbb{R}^m$, the set $D(\mathsf{C}, \vec{b}) = \{\vec{x} \in \mathbb{R}^n \mid \mathsf{C}\vec{x} \geq \vec{b}\}$ is called a* convex polytopic domain.

We interpret sorts $s \in S$ as convex domains $\mathcal{A}_s = D(\mathsf{C}_s, \vec{b}_s)$, where $\mathsf{C}_s \in \mathbb{R}^{m_s \times n_s}$ is an $m_s \times n_s$-matrix and $\vec{b}_s \in \mathbb{R}^{m_s}$. Thus, $\mathcal{A}_s \subseteq \mathbb{R}^{n_s}$.

Because of the subsort relation $s \leqslant s'$ $\mathcal{A}_s = D(\mathsf{C}_s, \vec{b}_s) \subseteq D(\mathsf{C}_{s'}, \vec{b}_{s'}) = \mathcal{A}_{s'}$ must hold. Also, $n_s = n_{s'}$ so the objects in both domains have the same dimension and the inclusion makes sense. In order to to ensure that all tuples in $\mathcal{A}_s$ belong to $\mathcal{A}_{s'}$, we use the following sufficient condition [10, Proposition 1]: If $\mathsf{C} = \mathsf{C}'$ and $\vec{b} \geq \vec{b}'$, then $D(\mathsf{C}, \vec{b}) \subseteq D(\mathsf{C}', \vec{b}')$.

2.2.2 Functions

As a generalization of a convex matrix interpretation, a *many-sorted convex matrix intepretation* for $f : s_1 \cdots s_k \rightarrow s$ is a linear expression $F_1\vec{x}_1 + \cdots + F_k\vec{x}_k + F_0$ such that (1) for all i, $1 \leq i \leq k$, $F_i \in \mathbb{R}^{n_s \times n_{s_i}}$ are $n_s \times n_{s_i}$-matrices, $\vec{x}_i \in \mathbb{R}^{n_{s_i}}$, (2) $F_0 \in \mathbb{R}^{n_s}$, and (3) it ranges on $D(\mathsf{C}_s, \vec{b}_s)$ whenever variables $\vec{x}_i$ take value on the corresponding domain $D(\mathsf{C}_{s_i}, \vec{b}_{s_i})$, i.e., that satisfies the following *algebraicity condition*:

$$\forall \vec{x}_1 \in \mathbb{R}^{n_{s_1}}, \ldots \forall \vec{x}_k \in \mathbb{R}^{n_{s_k}} \left(\bigwedge_{i=1}^{k} A_{s_i}\vec{x}_i \geq b_{s_i} \Rightarrow A_s(F_1\vec{x}_1 + \cdots + F_k\vec{x}_k + F_0) \geq b_s \right)$$

An $(S, \leq, \Sigma)$-algebra $\mathcal{A} = (\mathcal{A}, \Sigma_{\mathcal{A}})$ is obtained if $\mathcal{A} = \{D(A_s, b_s) \mid s \in S\}$, and each k-ary symbol $f \in \mathcal{F}$ is given a convex matrix interpretation $f_{\mathcal{A}}$ as above.

2.3 Conditional Polynomial Constraints

The use of convex domains and convex matrix interpretations amounts at dealing with *conditional constraints* of the form $\bigwedge_{i=1}^{n} e_i \geq d_i \Rightarrow e \geq d$ where e and e_i are *linear expressions* and d and d_i are *numbers*. Methods for solving conditional polynomial constraints have been described in early works [5], and propose a transformation of the conditional polynomial into a unconditional one. For polynomial constraints where the polynomial components are *linear*, we can apply Farkas' lemma.

Theorem 13 (Farkas's Lemma) *Let A be a matrix ant let b be a vector. There exists $y \geq 0$ with $Ay = b$ if and only if $b^T x \geq 0$ for each x with $A^T x \geq 0$.*

The Farkas' Lemma provides a simple way to transform the conditional constraint $A^T x \geq 0 \Rightarrow b^T x \geq 0$ into a constraint, solving the problem to find a *non-negative* vector $y \geq 0$ such that $Ay = b$. But we have to consider that a *pure* linear conditional constraints are not general enough for this purposes, so thats why we have to deal with constraints in the form $A^T x \geq b \Rightarrow c^T x \geq \beta$. We can use the *affine* form of Farkas' Lemma for this purpose:

Theorem 14 (Affine form of Farkas' Lemma) *Let $Ax \geq b$ be a linear system of k* inequalities *and n unknowns over the real numbers with non-empty solution set S and let $c \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$. Then, the following statements are equivalent:*

1. *$c^T x \geq \beta$ for all $x \in S$,*
2. *$\exists \lambda \in \mathbb{R}_0^k$ such that $c = A^T \lambda$ and $\lambda^T b \geq \beta$*

The affine conditional constraint $A\vec{x} \geq \vec{b} \Rightarrow c^T \vec{x} \geq \beta$ defined by $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$, $c \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ is equivalent to find a non-negative vector λ such that c is a linear non negative combination of the rows of A and and β is smaller than the corresponding linear combination of components of b.

3 Formal treatment of a verification problem

In this chapter we explain how to use the formal notions introduced in the previous chapter to handle a given verification problem expressed in order-sorted first-order logic. We follow [10, 11] and use the `WWV05-WEBSITE` specification to illustrate the process.

3.1 Defining the theory

With our running example, in the logic of Order-Sorted Term Rewriting Systems (OS-TRSs), with binary (overloaded) *predicates* $\rightarrow, \rightarrow^* \in \Pi_{ss}$ for each $s \in S$, the theory for an OS-TRS $\mathcal{R} = (S, \leq, \Sigma, R)$ with set of rules R (for instance, our running example) is obtained from the *schematic* inference rules in Figure 1.1 after *specializing* them according to the signature and rules in the program. Then, each specialized inference rule $\frac{B_1,\ldots,B_n}{A}$ becomes a universally quantified *implication* $B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

Taking our program example in Figure 1.2 and considering the inference rules in Figure 1.1, we can define the *theory* that corresponds to our specific program. In the following, we consider each generic inference rule and explain how to obtain the sentences in each case.

Reflexive Rule

In the logic of OS-TRSs, the Reflexive Inference Rule

$$\frac{}{t \rightarrow^* t}$$

means that, for each connected component $[s]$ in the hierachy of sorts defined by the ordering $\leq$ in S, each term t with sort in $[s]$ can be rewriten in one or more steps to the same term t. In our case, we have two connected components:

$$\begin{array}{rcl} [\texttt{User}] & = & \{\texttt{EventualUser}, \texttt{RegUser}, \texttt{User}\} \\ [\texttt{WebPage}] & = & \{\texttt{SecuerWebPage}, \texttt{WebPage}\} \end{array}$$

Therefore, this inference rule is specialized into *two* inference rules (one per connected component). In our running example, sorts `User` and `WepPage` can be taken as representatives of the corresponding connected components because they are supersorts of any other sort in the corresponding connected component. Thus, we obtain the following sentences representing the two inference rules.

$$\forall t : \texttt{User} \quad , \quad (t \to^* t) \tag{3.1}$$

$$\forall t : \texttt{WepPage} \quad , \quad (t \to^* t) \tag{3.2}$$

Transitive Rule

As in the case of reflexive inference, the Transitive Inference Rule

$$\frac{t \to t' \qquad t' \to^* u}{t \to^* u}$$

specializes into two different inference rules (one per connected component of sorts). We will obtain the following sentences:

$$\forall t, t', u : \texttt{User} \quad , \quad (t \to t' \wedge t' \to^* u \Rightarrow t \to^* u) \tag{3.3}$$

$$\forall t, t', u : \texttt{WebPage} \quad , \quad (t \to t' \wedge t' \to^* u \Rightarrow t \to^* u) \tag{3.4}$$

Congruence Rule

For the Congruence Inference Rule, we have to generate inference rules from the generic rule

$$\frac{t_i \to t'_i}{f(t_1, \ldots, t_i, \ldots, t_k) \to f(t_1, \ldots, t'_i, \ldots, t_k)}$$

for each function symbol f in the signature (with rank $f : s_1 \cdots s_k \to s$), and each argument i of the symbol (i.e., $i \in \{1, \ldots, k\}$). In our running example this becomes easier, because all functions are unary and we obtain a sentence per function symbol only:

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{wwv05}(u) \to \mathit{wwv05}(u')) \tag{3.5}$$

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{submission}(u) \to \mathit{submission}(u')) \tag{3.6}$$

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{sbmlink}(u) \to \mathit{sbmlink}(u')) \tag{3.7}$$

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{login}(u) \to \mathit{login}(u')) \tag{3.8}$$

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{vlogin}(u) \to \mathit{vlogin}(u')) \tag{3.9}$$

$$\forall u, u' : \texttt{User} \quad , \quad (u \to u' \Rightarrow \mathit{register}(u) \to \mathit{register}(u')) \tag{3.10}$$

$$\forall r, r' : \texttt{RegUser} \quad , \quad (r \to r' \Rightarrow \mathit{vlogin}(r) \to \mathit{vlogin}(r')) \tag{3.11}$$

$$\forall r, r' : \texttt{RegUser} \quad , \quad (r \to r' \Rightarrow \mathit{submit}(r) \to \mathit{submit}(r')) \tag{3.12}$$

Note that reduction steps with '$\to$' in the *antecedent* of the implications (e.g., $u \to u'$ or $r \to r'$) involve terms of sort `User` and `RegUser`, but reduction steps in the *consequent* involve terms of sort `WebPage` or `SecureWebPage`, i.e., they correspond to different overloadings of '$\to$'.

Replacement Rule

The Replacement Inference Rule

$$\frac{}{\ell \to r}$$

yields an instance for each rule $\ell \to r$ in the program $\mathcal{R}$, and we have to take into account the specific sort declaration given to the variables occurring in the program rules. In our running example, we obtain the following:

$$\begin{aligned}
\forall u : \texttt{User} \quad &, \quad (\mathit{wwv05}(u) \to \mathit{submission}(u)) &(3.13)\\
\forall u : \texttt{User} \quad &, \quad (\mathit{submission}(u) \to \mathit{sbmlink}(u)) &(3.14)\\
\forall u : \texttt{User} \quad &, \quad (\mathit{sbmlink}(u) \to \mathit{login}(u)) &(3.15)\\
\forall u : \texttt{User} \quad &, \quad (\mathit{sbmlink}(u) \to \mathit{register}(u)) &(3.16)\\
\forall u : \texttt{User} \quad &, \quad (\mathit{login}(u) \to \mathit{vlogin}(u)) &(3.17)\\
\forall r : \texttt{RegUser} \quad &, \quad (\mathit{vlogin}(r) \to \mathit{submit}(r)) &(3.18)
\end{aligned}$$

Note that the variable quantification in the last rule makes r to range on terms of sort `RegUser` only. This is consistent with the declaration of variable `R` in the program to have sort `RegUser` and its use in the definition of the last rule.

Note also that there is no use of predicate '$\to$' that *rewrites* a term of sort `User`. From a logic point of view, we can say that no relation named '$\to$' is established between terms of sort `User` (or that the *overloading* of $\to$ for `User` is going to be *empty*).

Specific conditions of the analysis

Besides the definition of the specification and its semantics as given by the inference rules of the OS-TRS logic, we may add further sentences representing the property to be verified. In our current example, we can express this property using OS-FOL sentences as well:

$$\begin{aligned}
\forall \texttt{u} : \texttt{User}, \texttt{wwv05(u)} \to^* \texttt{submit(u)} \quad &\Rightarrow \quad \neg(\texttt{u} : \texttt{EventualUser}) &(3.19)\\
\forall \texttt{u} : \texttt{User}, \texttt{wwv05(u)} \to^* \texttt{submit(u)} \quad &\Rightarrow \quad \texttt{u} : \texttt{RegUser} &(3.20)
\end{aligned}$$

3.2 Defining the model using convex domains

3.2.1 Domains for sorts

In our running example, a convex domain matrix $\mathsf{C}_s \in \mathbb{R}^{m_s \times n_s}$ and vector $\vec{b}_s \in \mathbb{R}^{n_s}$ is associated to each sort s, so that the domain $\mathcal{A}_s$ given to a sort s is the convex domain generated by C_s and $\vec{b}_s$, i.e., $\mathcal{A}_s = D(\mathsf{C}_s, \vec{b}_s)$. Furthermore, we let $m_s = 2$ and $n_s = 1$ for all sorts $s \in S$.

The dimension of the matrix C that is used to define the convex domain must be the *same* for all sorts s in the same connected component (see Section 2.2.1). Remember that we have two connected components:

$$\begin{aligned}[\texttt{User}] &= \{\texttt{EventualUser}, \texttt{RegUser}, \texttt{User}\}\\ [\texttt{WebPage}] &= \{\texttt{SecuerWebPage}, \texttt{WebPage}\}\end{aligned}$$

In this particular case, we will define two parametric 2×1-matrices by $\mathsf{C}_{\texttt{User}} = (C_1^u, C_2^u)^T$ and $\mathsf{C}_{\texttt{WebPage}} = (C_1^w, C_2^w)^T$.

The parametric vectors for $s \in S$ are given by $\vec{b}_s = (b_1^s, b_2^s)^T$ and can be different for each sort. According to Section 2.2.1, this leads to the following initial constraints (the name of the sorts have been shortened to make the reading easier):

$$b_1^{\texttt{EU}} \geq b_1^{\texttt{U}} \wedge b_2^{\texttt{EU}} \geq b_2^{\texttt{U}} \wedge b_1^{\texttt{RU}} \geq b_1^{\texttt{U}} \wedge b_2^{\texttt{RU}} \geq b_2^{\texttt{U}} \wedge b_1^{\texttt{SWP}} \geq b_1^{\texttt{WP}} \wedge b_2^{\texttt{SWP}} \geq b_2^{\texttt{WP}} \tag{3.21}$$

that guarantee that the subsort hierarchy is translated as *subset inclusion* between the convex domains associated to the different sorts.

3.2.2 Interpretation of function symbols

Since in our running example, we define $n_s = 1$ for all $s \in \mathcal{S}$, we give *parametric interpretations* to each $f \in \Sigma$ as follows:

$$([wwv05](x) = w_1x + w_0) \tag{3.22}$$
$$([submission](x) = m_1x + m_0) \tag{3.23}$$
$$([sbmlink](x) = s_1x + s_0) \tag{3.24}$$
$$([login](x) = l_1x + l_0) \tag{3.25}$$
$$([register](x) = r_1x + r_0) \tag{3.26}$$
$$([vlogin](x) = v_1x + v_0) \tag{3.27}$$
$$([submit](x) = t_1x + t_0) \tag{3.28}$$

where w_1, w_0, m_1, etc., are *parameters* which will be given appropriate values after a constraint solving processs (see Chapter 5). The necessary *algebraicity conditions* for these functions are given as follows (with x universally quantified in all formulae):

$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(l_1x + l_0) \geq b_1^{\texttt{WP}} \wedge C_2^w(l_1x + l_0) \geq b_2^{\texttt{WP}} \tag{3.29}$$
$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(r_1x + r_0) \geq b_1^{\texttt{WP}} \wedge C_2^w(r_1x + r_0) \geq b_2^{\texttt{WP}} \tag{3.30}$$
$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(s_1x + s_0) \geq b_1^{\texttt{WP}} \wedge C_2^w(s_1x + s_0) \geq b_2^{\texttt{WP}} \tag{3.31}$$
$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(m_1x + m_0) \geq b_1^{\texttt{WP}} \wedge C_2^w(m_1x + m_0) \geq b_2^{\texttt{WP}} \tag{3.32}$$
$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(w_1x + w_0) \geq b_1^{\texttt{WP}} \wedge C_2^w(w_1x + w_0) \geq b_2^{\texttt{WP}} \tag{3.33}$$
$$C_1^u x \geq b_1^{\texttt{U}} \wedge C_2^u x \geq b_2^{\texttt{U}} \Rightarrow C_1^w(v_1x + v_0) \geq b_1^{\texttt{SWP}} \wedge C_2^w(v_1x + v_0) \geq b_2^{\texttt{SWP}} \tag{3.34}$$
$$C_1^u x \geq b_1^{\texttt{RU}} \wedge C_2^u x \geq b_2^{\texttt{RU}} \Rightarrow C_1^w(t_1x + t_0) \geq b_1^{\texttt{SWP}} \wedge C_2^w(t_1x + t_0) \geq b_2^{\texttt{SWP}} \tag{3.35}$$

3.2.3 Interpretation of predicate symbols

In our example, for the translation of the universally quantified rules of the OS-TRS logic into arithmetic formulae, we will have the predicates $\rightarrow, \rightarrow^* \in \Pi_{ss}$ interpreted as $\geq$ (the usual ordering on numbers) for all $s \in S$. For instance, a formula $s \rightarrow^* t$ will be translated as $s \geq t$.

3.2.4 Derived theory

The *derived theory* which is obtained from the theory in Section 3.1 by using the previous interpretation of sorts and symbols will be:

1. Instances of the Reflexivity Rule (Re) with t universally quantified (like prior text, the name of the sorts have been shortenned to make the reading easier):

$$C_1^w t \geq b_1^{\mathtt{WP}} \wedge C_2^w t \geq b_2^{\mathtt{WP}} \Rightarrow t \geq t \tag{3.36}$$
$$C_1^u t \geq b_1^{\mathtt{U}} \wedge C_2^u t \geq b_2^{\mathtt{U}} \Rightarrow t \geq t \tag{3.37}$$

2. Instances of the Transitivity Rule (T) where we write (for instance) $t \in \mathcal{A}_{\mathtt{WP}}$ instead of $C_1^w t \geq b_1^{\mathtt{WP}} \wedge C_2^w t \geq b_2^{\mathtt{WP}}$, which formalizes the real membership condition for the domain $\mathcal{A}_{\mathtt{WP}}$ of sort `WebPage`:

$$t \in \mathcal{A}_{\mathtt{WP}} \wedge t' \in \mathcal{A}_{\mathtt{WP}} \wedge u \in \mathcal{A}_{\mathtt{WP}} \wedge t \geq t' \wedge t' \geq u \Rightarrow t \geq u \tag{3.38}$$
$$t \in \mathcal{A}_{\mathtt{U}} \wedge t' \in \mathcal{A}_{\mathtt{U}} \wedge u \in \mathcal{A}_{\mathtt{U}} \wedge t \geq t' \wedge t' \geq u \Rightarrow t \geq u \tag{3.39}$$

3. Instances of the Congruence Rule (C) were not added, because all function symbols f have argument sort `User` or `RegUser` and the rewrite relation associated to $\rightarrow \in \Pi_{\mathtt{User\,User}}$ is *empty.*

4. Instances of the Replacement Rule (Re) where the variables u and r are universally quantified ranging on $\mathcal{A}_{\mathtt{U}}$ and $\mathcal{A}_{\mathtt{RU}}$, respectively:

$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \Rightarrow w_1 u + w_0 \geq m_1 u + m_0 \tag{3.40}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \Rightarrow m_1 u + m_0 \geq s_1 u + s_0 \tag{3.41}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \Rightarrow s_1 u + s_0 \geq l_1 u + l_0 \tag{3.42}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \Rightarrow s_1 u + s_0 \geq r_1 u + r_0 \tag{3.43}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \Rightarrow l_1 u + l_0 \geq v_1 u + v_0 \tag{3.44}$$
$$C_1^u r \geq b_1^{\mathtt{RU}} \wedge C_2^u r \geq b_2^{\mathtt{RU}} \Rightarrow v_1 r + v_0 \geq t_1 r + t_0 \tag{3.45}$$

3.2.5 Specific Conditions for the Analysis

As a final step in the Convex Domain analysis, its needed to generate an interpreted version of the goals defined in 3.19 and 3.20, where we assume that u is universally quantified:

$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad \neg(C_1^u u \geq b_1^{\mathtt{EU}} \wedge C_2^u u \geq b_2^{\mathtt{EU}})$$

$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad C_1^u u \geq b_1^{\mathtt{RU}} \wedge C_2^u u \geq b_2^{\mathtt{RU}}$$

which is equivalent to this, since the intended meaning of $\geq$ is the *total* order on numbers:

$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad b_1^{\mathtt{EU}} > C_1^u u \vee b_2^{\mathtt{EU}} > C_2^u u \tag{3.46}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad C_1^u u \geq b_1^{\mathtt{RU}} \wedge C_2^u u \geq b_2^{\mathtt{RU}} \tag{3.47}$$

and then to the following sentences, finally in the required form for dealing with them using Farkas' Lemma:

$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \wedge C_1^u u \geq b_1^{\mathtt{EU}} \quad \Rightarrow \quad b_2^{\mathtt{EU}} \geq C_2^u u + \delta \tag{3.48}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad C_1^u u \geq b_1^{\mathtt{RU}} \tag{3.49}$$
$$C_1^u u \geq b_1^{\mathtt{U}} \wedge C_2^u u \geq b_2^{\mathtt{U}} \wedge w_1 u + w_0 \geq t_1 u + t_0 \quad \Rightarrow \quad C_2^u u \geq b_2^{\mathtt{RU}} \tag{3.50}$$

for some $\delta > 0$.

Also, according to [10], we need to further ensure that the unique homomorphism

$$h_{\mathtt{RegUser}} : \mathcal{T}_\Sigma(\mathcal{X})_{\mathtt{RegUser}} \to \mathcal{A}_{\mathtt{RegUser}}$$

is surjective, which means that for every value in $\mathcal{A}_{\mathtt{RegUser}}$ exists a value in $h_{\mathtt{RegUser}} : \mathcal{T}_\Sigma(\mathcal{X})_{\mathtt{RegUser}}$. Since $\mathcal{T}_\Sigma(\mathcal{X})$ is not empty (due to the presence of a variable `R` of sort `RegUser`), this is easily achieved if $\mathcal{A}_{\mathtt{RegUser}}$ is a singleton.

For our convex domains, we have the following sufficient condition.

Proposition 15 *A convex domain $D(\mathsf{C}, \vec{b})$ with $\mathsf{C} = (c_1, c_2)^T$ and $\vec{b} = (b_1, b_2)^T$ is a singleton if $c_1 = -c_2 \neq 0$ and $b_1 = -b_2$.*

We can use this sufficient condition in our specific problem by further requiring

$$C_1^u \neq 0 \wedge C_1^u = -C_2^u \wedge b_1^{\mathtt{RU}} = -b_2^{\mathtt{RU}} \tag{3.51}$$

4
The AGES Tool

The *Automatic GEneration of logical modelS* (AGES) tool tries to generate a model for an order-sorted first-order *theory* using the methodology presented in this thesis. Such theory consists of a *program specification* in Maude syntax and a set of logic formulae that express a property of interest. The tool is written in Haskell and is based on the framework created for MU-TERM. We also envisage the integration of the model generation implemented in AGES into MU-TERM to add new features to that tool in the future. In this chapter, we detail the process accomplished inside the tool to convert a program specification and a logic formula into a model. The tool has two user interfaces: A user-friendly web user interface for quick and easy testing, and a console application that can be used to automate analysis processes in large batch of tests. Internally, both applications use the same model-generation engine, and only the way for the user to receive the outcome of the tool changes.

4.1 Internal structure and work flow

Internally, the application executes the following processes, in the listed order:

1. Parse the command line arguments, or the contents of web form text-boxes, depending on the case.
2. Read the input data, currently a Maude specification and a goal (a set of logic formulae), and optional parameters.
3. Parse the program specification into an OS-TRS data type.
4. Parse the goal into a First-Order Logic Formula.
5. For each *Sort* in the program, generate the *Parametric Convex Domain Matrices and Vectors*.
6. For each symbol in the signature, generate a linear polynomial interpretation, taking into account the convex domains associated to the different sorts.
7. From the input program specification and the *Inference Rules* in Figure 1.1, generate a new set of First-Order Logic Formulae.

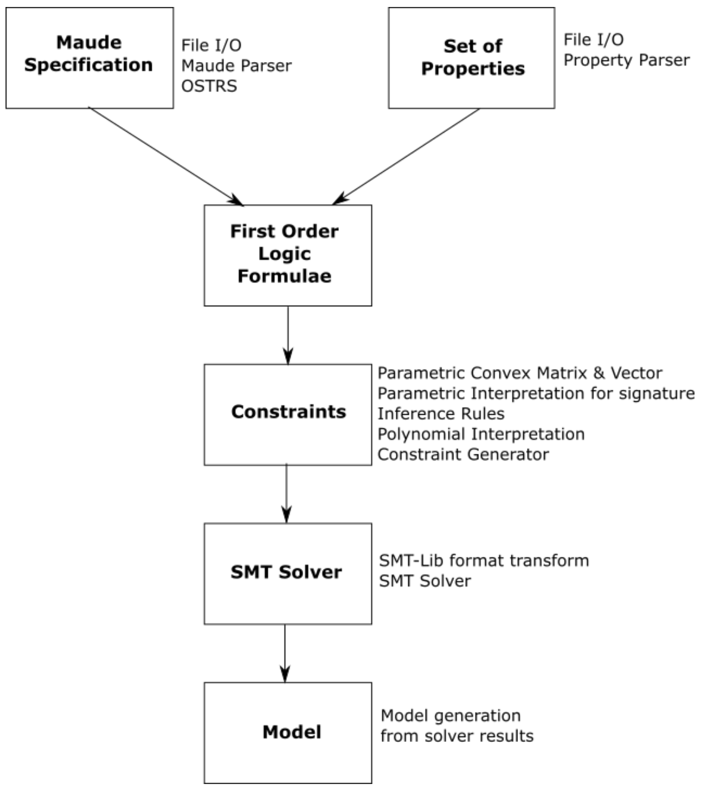


Figure 4.1: AGES Workflow

8. From the set of First-Order Formulae generated from the input program specification and the input goal, generate a set of conditional arithmetic constraints.

9. Remove *Conditional Implications* from the constraints, aplying Farkas's theorem.

10. Send the generated constraints to a *constraint solver*.

11. Format and show the resulting model.

As we can see, the model generation is based on several transformations stepwise applied to the input theory. In fact, after reading the input data (a program in MAUDE format, see section 4.2 for further information about the syntax, a goal and the working parameters), the first transformation is converting the MAUDE program, and the logical formulae in the goal component to be tested into a set of logic formulae, applying the inference rules in Figure 1.1. By extracting the *Sorts* information from the OS-TRS, we can generate the convex domain matrices (see Section 3.2), which will be used to apply the convex domain interpretation to all variables contained in the inference rules created with the last transformation.

Once a set of formulae has been obtained, we transform it into conditional arithmetic constraints, using polynomial interpretations. The polynomial interpretation transformation, essentially uses the clasical method described, for example, in [3] and [12] and generates a set of polynomial conditional constraints.

Remark 16 Currently, *available* interpretations for predicates $\to$ and $\to^*$ are limited to $>$, $\geq$ with the usual arithmetic interpretations, and $=$ (equality). ■

The conditional implication removal process uses a direct application of Farkas's theorem, see Section 2.3.

Finally, a simplification process is made to all constraints in order to reduce the size of the final constraint that will be send to the external SMT solver. In our case, we use *Barcelogics SMT Solver*, which can be found at:

http://www.cs.upc.edu/~albert/nonlinear.html).

4.2 Input data and data format

The input data for AGES tool, consists of two separated but mandatory inputs:

- A MAUDE specification, that represents the input theory to be analyzed.
- A logical property to be tested, which is given as a worked set of formulae. Together they form an OS-FOL theory.

Also, there are two optional parameters that can be used. We will describe them later in this chapter.

4.2.1 MAUDE program syntax

The MAUDE syntax is very simple, but also very powerful and expressive. In this section we recall the basic knowledge of Reweite Systems, whch is required to write a program for our AGES tool. Full details about the syntax of MAUDE can be found here http://maude.cs.uiuc.edu/maude2-manual/html/maude-manualch3.html.

Module Definition

First, we need to specify our theory as a new *system module*. This can be done using the reserved word `mod`, and assigning a name to the module. The module name must be written in uppercase characters, as in the following example:

Example 17 (System module definition for a MAUDE program)

```
mod WWV05-WEBSITE is
....
endm
```

We use *system modules* because our specification represents state changes by means of *rewrite rules*, *functional modules* are used to define algebraic specifications by means of equations and are defined with the reserved word *fmod*.

Sorts Definition

We need to define the poset of sorts of the specification. The definition of *sorts* is part of the *module signature*. In this case we use the reserved word `sorts`, followed by the name of all the sorts required, as we can see in the followng example, which includes the module definition:

Example 18 (Sorts definition for a MAUDE program)

```
mod WWV05-WEBSITE is

sorts EventualUser RegUser User WebPage SecureWebPage .

....
endm
```

Subsort Relations

We can use the `subsort` reserved word, and MAUDE's reserved operator '<', to define the subsort relationships between sorts of the module. The following example shows the subsort relationship between the sorts of our theory, where there are *two* subsort relations, one for the *User supersort*, and another one for the *WebPage supersort*:

Example 19 (Sorts definition for a MAUDE program)

```
mod WWV05-WEBSITE is

sorts EventualUser RegUser User WebPage SecureWebPage .

subsorts RegUser EventualUser < User .
subsorts SecureWebPage < WebPage .
....
endm
```

Operators

The *operators* provide names (i.e. function symbols) for the operations that will act upon the data. That allows us to build terms refered to such data. In this case, we define the function names f, and their input and output *sorts* as a *rank* written `op` f `:` s_1 s_2 s_k `->` s where s_1, s_2,...,s_k and s are sorts. Notice in the next example, that we can use the reserved words `op` for a single function name, and `ops` for multiple function names sharing the same rank.

Example 20 (Operators definition for a MAUDE program)

```
mod WWV05-WEBSITE is

sorts EventualUser RegUser User WebPage SecureWebPage .

subsorts RegUser EventualUser < User .
subsorts SecureWebPage < WebPage .

ops login register sbmlink submission wwv05 : User -> WebPage .
op vlogin : User -> SecureWebPage .
op submit : RegUser -> SecureWebPage .

....
endm
```

Variables

Before defining the rules in our MAUDE specification, we can define the variables used later in the rules. These variables also have a *sort* whose name must be included in the definition. In MAUDE we use the reserved word `var` and the syntax `var` v `:` S for a variable identifier v and sort S. In the following example, we define two variables that will be used in our rules: `R` of sort `RegUser`, and `U` of sort `User`.

Example 21 (Operators definition for a MAUDE program)

```
mod WWV05-WEBSITE is

sorts EventualUser RegUser User WebPage SecureWebPage .

subsorts RegUser EventualUser < User .
subsorts SecureWebPage < WebPage .

ops login register sbmlink submission wwv05 : User -> WebPage .
op vlogin : User -> SecureWebPage .
op submit : RegUser -> SecureWebPage .
```

```
var R : RegUser .
var U : User .
....
endm
```

Rules

Finally, the MAUDE specification can include *rewrite rules*. This completes our program specification. Each rule is included by means of the `rl` reserved word, followed by l `=>` r for terms l and r. All functions and variables used in the rules, must be previously declared in the specification. As for our example program, we have the following:

Example 22 (Program Rules defined for a MAUDE program)

```
mod WWV05-WEBSITE is

sorts EventualUser RegUser User WebPage SecureWebPage .

subsorts RegUser EventualUser < User .
subsorts SecureWebPage < WebPage .

ops login register sbmlink submission wwv05 : User -> WebPage .
op vlogin : User -> SecureWebPage .
op submit : RegUser -> SecureWebPage .

var R : RegUser .
var U : User .

rl wwv05(U) => submission(U) .
rl submission(U) => sbmlink(U) .
rl sbmlink(U) => login(U) .
rl sbmlink(U) => register(U) .
rl login(U) => vlogin(U) .
rl vlogin(U) => submit(R) .
rl login(U) => vlogin(U) .
rl vlogin(R) => submit(R) .

endm
```

4.2.2 Goal Definition and Format

Besides the MAUDE specification of our program, in order to generate a model using the AGES tool, we need to define a set of *goals* to be tested, like the ones defined in the example formulae 3.19 and 3.20. These formulae also need to be defined in a proper format, before being able to handle them in the tool.

For defining formulae, the allowed predicates are $\to$ and $\to^*$ (which can be freely interpreted when defining models). We can also use *True* and *False*, $=$ (the equality predicate), and connectives $\wedge$, $\vee$, $\neg$ and $\Rightarrow$ (this symbol represents the implicaton), all of them with the standard meaning. The syntax for the predicate symbols defined for the goal definition can be found in Table .

Predicates	AGES' syntax
$\to$	`->`
$\to^*$	`->*`
$=$	`=`
$:$	`:`

Logical symbol	AGES' syntax
$\Rightarrow$	`=>`
True	`true`
False	`false`
$\neg$	`~`
$\wedge$	`/\`
$\vee$	`\/`

Table 4.1: Goal syntax

Because of a tool restriction, we need to define the *Sort* of each variable used inside the formula and all variables are *universally quantified* by default. We provide some examples to ilustrate the syntax of the goal formulae. The following example 23 shows the use of the predicate symbols.

Example 23 (Predicate use and syntax)

- The formula $True \Rightarrow \neg False$ can be defined as:

```
true => ~(false)
```

- The formula $True \vee False \Rightarrow True$ can be defined as:

```
true \/ false => (true)
```

- The formula $(foo(x) \wedge bar(y)) \vee foo(y) \Rightarrow \neg(True)$ can be defined as:

```
(foo(x) /\ bar(y)) \/ foo(y) => ~(true)
```

- The formula $(foo(x) \to bar(x)) \wedge (foo(y) \to^* bar(y)) \Rightarrow False$ can be defined as:

  ```
  (foo(x) -> bar(x)) /\ (foo(y) -> bar(y)) => false
  ```

The following example takes a formula from the running example, and shows how should be written to by handled by the tool.

Example 24 (Goal definition in AGES for the running example)
The formula $\forall \mathtt{u} : \mathtt{User}, \mathtt{wwv05(u)} \to^* \mathtt{submit(u)} \Rightarrow \neg(\mathtt{u} : \mathtt{EventualUser})$ can be defined as follows:

```
wwv05(U:User) ->* submit(U:User) => ~(U:EventualUser)
```

4.3 A short user manual

The AGES (Automatic GEneration of logical modelS) tool has two main components:

- A web based application, with an user friendly interface.
- A console based tool, suited for large batch process analysis.

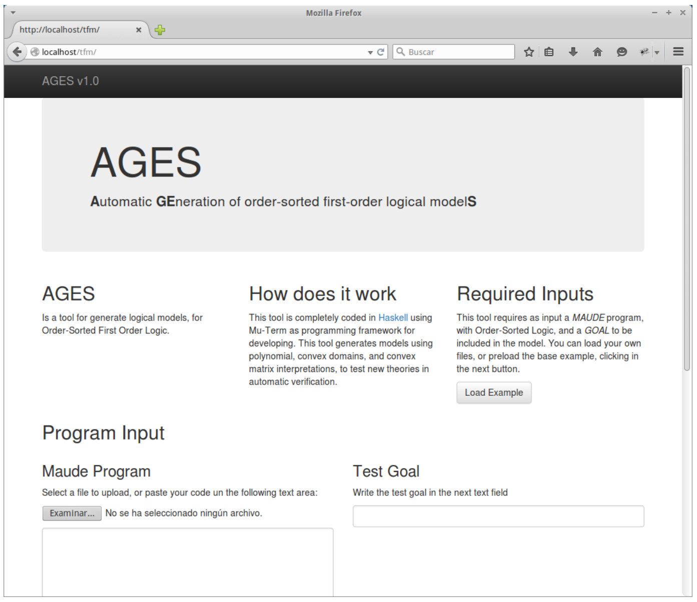
Both tools, share the same core code. In the following we describe their functionality.

4.3.1 Web Based Application

The web based application provides an easy, friendly interface to the model generator. In order to use this application, the end user only needs a *HTML5* compatible browser. The main page of the tool is displayed on Figure 4.2.

In the main web page, there are four fields, that need to be filled, before generating the model:

- *Program Input*: This field will accept the MAUDE program used to generate the model. The program can be pasted from a text file, or uploaded using the *Browse* button above the field. This field is required.
- *Goal*: This field allows to add a goal to the model.The goal definition format can be found in Section 4.2.2
- *Interpretation*: The user can select the polynomial interpretation for generating the model. Currently only *Linear* interpretations are allowed.


Mozilla Firefox

http://localhost/tfm/

localhost/tfm/

Buscar

AGES v1.0

AGES

Automatic **GE**neration of order-sorted first-order logical model**S**

AGES

Is a tool for generate logical models, for Order-Sorted First Order Logic.

How does it work

This tool is completely coded in Haskell using Mu-Term as programming framework for developing. This tool generates models using polynomial, convex domains, and convex matrix interpretations, to test new theories in automatic verification.

Required Inputs

This tool requires as input a *MAUDE* program, with Order-Sorted Logic, and a *GOAL* to be included in the model. You can load your own files, or preload the base example, clicking in the next button.

Load Example

Program Input

Maude Program

Select a file to upload, or paste your code un the following text area:

Examinar... No se ha seleccionado ningún archivo.

Test Goal

Write the test goal in the next text field

Figure 4.2: AGES Web App Main Page

- *Matrix Size*: This field allows to change the dimension of the coefficients in the polynomial interpretation of the symbols in the signature. In the current version of the tool, the value is fixed to 1.

When all the fields are properly entered, we can press the *Generate* button, and start the model generation process. In a short time (depending on the complexity of the program, and if it has solution or not).

4.3.2 Console Application

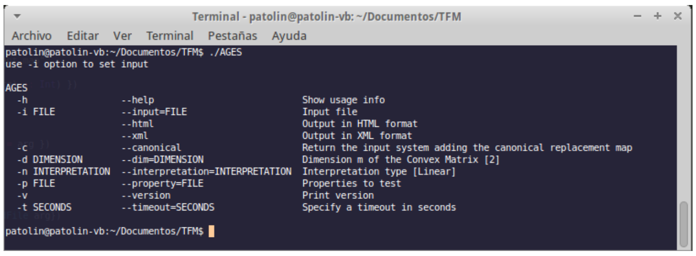
The console application has the same functionality as the web application, but it is intended to be used in an *automated* environment; for example, for processing batches of programs and goals to perform a long unattended run test.

When the AGES command is invoked from a console terminal, if no parameters are included, the command line help is displayed, as we can see in Figure 4.3.

In order to use this version of the tool, we need two additional text files for each program we want to model:

- A file with extension *.maude*, that contains the MAUDE program.
- A file with extension *.prop* that contains the *goal* to be tested.

Using our running example, with the file *wwv05.maude* containing the MAUDE program, and the file *prop2.prop* containing the goal to test, we can use the following command to generate a model.


Terminal - patolin@patolin-vb: ~/Documentos/TFM
Archivo Editar Ver Terminal Pestañas Ayuda

```
patolin@patolin-vb:~/Documentos/TFM$ ./AGES
use -i option to set input

AGES
  -h                  --help                            Show usage info
  -i FILE             --input=FILE                      Input file
                      --html                            Output in HTML format
                      --xml                             Output in XML format
  -c                  --canonical                       Return the input system adding the canonical replacement map
  -d DIMENSION        --dim=DIMENSION                   Dimension m of the Convex Matrix [2]
  -n INTERPRETATION   --interpretation=INTERPRETATION   Interpretation type [Linear]
  -p FILE             --property=FILE                   Properties to test
  -v                  --version                         Print version
  -t SECONDS          --timeout=SECONDS                 Specify a timeout in seconds

patolin@patolin-vb:~/Documentos/TFM$
```

Figure 4.3: AGES console application help

```
./AGES -i wwv05.maude -p prop2.prop -n Linear -d 1
```

Where the following parameters are used:

- *-i*: Defines the file that contains the *.maude* MAUDE program.
- *-p*: Defines the file that contains the *.prop* goal to test.
- *-n*: Sets the polynomial interpretation type. In this version, only the *Linear* interpretation is available, and that is the default value.
- *-d*: Sets the dimension of the coefficients in the polynomial interpretation. The default value is *1*

5

Practical Implementation

This chapter describes the implementation of AGES, from the setup of the development environment to the issues of Haskell encoding. We also discuss the new created modules and their relationship with the current MU-TERM implementation.

5.1 Development Environment

In order to develop the tool, we need to setup the environment with the required software tools. For setup ease, we use Linux as operating system, with the Ubuntu 14.10 “Utopic Unicorn” distribution. In our specific case, we use the 32 bits version, because we are installing it over a VirtualBox Virtual Machine. Virtualization brings a sandboxed environment to work, and the portability of the virtual machine to a production environment, easier than making a whole setup from scratch.

5.1.1 Haskell Compiler

The code of AGES uses some of the libraries originally developed for MU-TERM. For this reason, we need to use the Glasgow Haskell Compiler (GHC), Version 7.8.3. With Ubuntu (or any Debian based) distribution, the installation of the GHC compiler is a trivial task. We need to use the following command from a terminal console:

```
sudo apt-get install ghc
```

Once the installation process finishes, we can test the GHC setup with the command:

```
ghc --version
```

to obtain an output like

```
The Glorious Glasgow Haskell Compilation System, version 7.8.3
```

5.1.2 Cabal Installer

Cabal is a system for building and packing Haskell libraries. Since MU-TERM uses it to manage all its modules, its installation is mandatory. To setup cabal, we use the following command, from a console terminal:

```
sudo apt-get cabal-install
```

After its completion, the setup process can be tested with this command:

```
cabal --version
```

If the installation succeds, it will return an output like:

```
cabal-install version 1.20.0.3
using version 1.20.0.2 of the Cabal library
```

Also, as a good practice, updating the cabal package list is recommended:

```
cabal update
```

5.1.3 MU-TERM Framework

MU-TERM Framework is a package that provides an abstract divide-and-conquer framework based on a strategy. It was created to be shared with other tools that use a divide-and-conquer framework, as Narradar[1].

Since the MU-TERM framework is necessary to compile MU-TERM, we need to install it by using cabal. For this purpose, we have to clone the repository in our local computer with the following commands:

```
git clone https://github.com/pepeiborra/muterm-framework.git
```

and then, using cabal, we install the package with the following commands:

```
cd muterm-framework
cabal configure
cabal install
```

If no error shows up during the installation, the MU-TERM framework package is ready for use in any Haskell program.

5.1.4 MU-TERM Initial Setup

In order to use MU-TERM as the base code of our project, it's necessary to install the latest version of the code, and compile it properly. If the previous steps do not display any errors, we can download and decompress the MU-TERM source code, and execute the following commands in the home directory of the code:

[1]http://safe-tools.dsic.upv.es/narradar/

```
cabal configure
cabal install
ghc Main.hs
```

If the code compiles without errors, we can start to coding our own project, using MU-TERM as the base platform.

5.2 Implementing AGES using MU-TERM

In the last section, we described the steps to get our development environment ready. Now, we need to define the proper features of our project, and start defining the code requirements for each new module to be developed.

- The existing Maude *parser* must be validated, so we can prove that it can handle Order-Sorted Term Rewriting Systems.
- The *command line interface* needs to include an option to set the name of the Maude file with the program specification, the name of the PROP file, that contains the property to be tested, and some extra parameters, like the type of interpretation and the size of the matrix defined for the convex domain analysis.
- The *property* to be tested needs to be loaded and parsed. Thus, a new module is needed to handle this task.
- Taking the Maude specification, and the *property* to be tested, the tool must generate the logic formulae, according to the inference rules in Figure 1.1.
- Taking the generated formulae, we need to generate the constraints.
- Finally, we transform the constraints using Farkas and send the to a SMT solver to obtain a model of the input theory and goal.

In the next subsections we describe each step in detail.

5.2.1 Maude Parser Module

The Maude parser module contained in MU-TERM can be used to process a .maude file that contains a source specification. In particular we can parse an Order-Sorted specification. The Haskell module `Parser.MAUDE.Parser` exports the function `parseMAUDE` that has the following type:

```
parseMAUDE :: String -> Either ParseError (MProblem info)
```

This function takes a string with the .maude file contents and returns either a `ParseError` (if the program could not be parsed, an error with the line number an the expected code is showed), or an `MProblem` type variable, that contains all the Maude program information. In this case, the data `MProbleminfo` will have the type:

```
MMaudeProblem   :: Problem MaudeProblem   InMaude  -> MProblem info
```

where the `InMaude` type represents a TRS with Order-Sorted Associated theory.

```
type InMaude = OSTRS
(TRS (IdFOS (IdFAvC IdFTRS))
(IdVOS IdVTRS)
(Term (IdFOS (IdFAvC IdFTRS)) (IdVOS IdVTRS))
)
```

Also, for the `OSTRS`, `TRS`, `IdFOS`, `IdVOS`, and `IdVTRS` types, we have the following definitions

```
-- | Order-sorted term rewriting system information
data OSTRS trs = OSTRS
{ ostrsprev      :: trs         -- \^ Previous info
, ossignature    :: OSSignature -- \^ Order-sorted signature
} deriving Show

-- | Parametrized term rewriting system. It is formed by a signature,
-- a set of variables and a set of rules.
data TRS a b c
= TRS { trsSignature  :: Signature a
, trsVariables  :: Variables b
, trsRules      :: Set (Rule c)
, trsLabel      :: TRSLabel
} deriving (Eq,Show)

-- | Order-sorted function symbol
data IdFOS id = IdFOS
{ osfprev        :: id          -- \^ Previous info
, osfsorts       :: [FSort]     -- \^ List of possible sorts
} deriving Show

-- | Associative-commutative function symbol
data IdFAvC id = IdFAvC
{ avcfprev        :: id          -- \^ Previous info
, avc             :: AvC         -- \^ AvC symbol
} deriving Show

-- | Term rewriting system function symbol
data IdFTRS = IdFTRS
{ fid            :: Int         -- \^ Identifier
, fname          :: String      -- \^ Name of the function symbol
, arity          :: Int         -- \^ Arity of the symbol
} deriving Show

-- | Order-sorted variable information
data IdVOS id = IdVOS
```

```
{ osvprev         :: id          -- \^ Previous info
, vsort           :: Sort        -- \^ Variable sort
} deriving Show

-- | Variable information
data IdVTRS = IdVTRS
{ vid            :: Int          -- \^ Identifier
, vname          :: Maybe String -- \^ Name of the variable
} deriving Show
```

5.2.2 Command Line Interface Parameters

As mentioned before, the command line of the tool must accept parameters that are not allowed in the MU-TERM command line. The new parameters to be included are:

- The name of the file that contains the property to be tested.
- The type of the interpretation that will be used in the polynomials generation.
- The dimension of the matrix used for the *polynomial interpretations.*

These new parameters require a modification of the code that handles the command line. First of all, we need to create four new options for the `Opt` type:

- `optPropInput`: for the property file contents.
- `optPropName`: for the property file name.
- `optDim`: for the dimension m of the coefficients.
- `optInterpretation`: for the type of interpretation that will be used.

Thus, the *main* function needs to include these types, as follows:

```
main :: IO ()
main = do (opts, _) <- parseOptions
let Opt { optName            = name
, optInput           = input
, optOutput          = output
, optFormat          = format
, optTimeout         = timeout
, optCanonical       = canonical
, optTest            = temporal
, optPropName        = propName
, optPropInput       = propInput
, optDim             = dim
, optInterpretation  = interp
} = opts
```

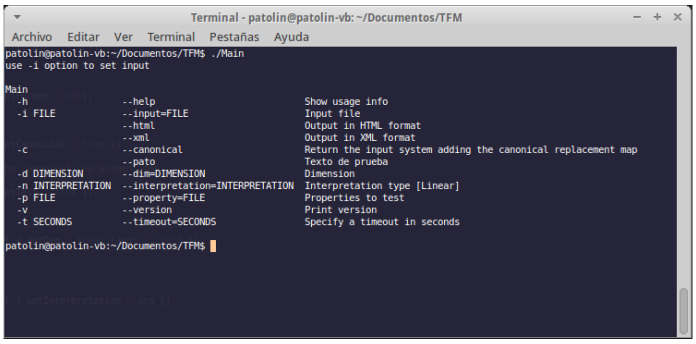
Besides this new definition, the tool has to obtain this information from the command line arguments. This information is handled by module `Interface.CLI`, where first of all, the data type `Opt` needs to be set as in the main function, so, it's defined as follows:

```
data Opt = Opt {
optName           :: String            -- ^ Input path
, optInput          :: IO String         -- ^ Input file
, optOutput         :: Output -> SomeInfo PrettyInfo
-> Solution (Proof PrettyInfo [] ())
-> IO ()   -- ^ Output formatted
, optFormat         :: Output            -- ^ Output format
, optTimeout        :: Maybe Int         -- ^ Timeout
, optCanonical      :: Bool              -- ^ Add canonical replaing map
, optPropName       :: String
, optPropInput      :: IO String
, optDim            :: Int
, optInterpretation :: String
}
```

Also, we define default values for the arguments (just in case that any argument is not present in the command line) and parse the command line arguments, using the following functions:

```
-- | Default parameters
startOpt :: Opt
startOpt = Opt {
optName           = "foo"
, optInput          = exitErrorHelp "use -i option to set input"
-- a simple way to handle mandatory flags
, optOutput         = \format problem someSol
-> putStr . show . pPrint $ someSol
, optFormat         = Plain
, optTimeout        = Nothing
, optCanonical      = False
, optPropName       = "propertyName"
, optPropInput      = exitErrorHelp "use -p option to set property input"
, optDim            = 1
, optInterpretation = "Linear"
}

-- | Command line options
options :: [OptDescr (Opt -> IO Opt)]
options =  [
Option "h" ["help"]
(NoArg (\opt -> exitHelp))
"Show usage info"
, Option "i" ["input"]
(ReqArg (\arg opt -> do return opt { optName = arg
, optInput = readFile arg})
"FILE"
)
"Input file"
, Option "" ["html"]
(NoArg (\opt -> do return opt { optFormat = HTML})
)
"Output in HTML format"
, Option "" ["xml"]
(NoArg (\opt -> do return opt { optFormat = XML})
)
"Output in XML format"
, Option "c" ["canonical"]
(NoArg (\opt -> do return opt { optCanonical = True })
)
"Return the input system adding the canonical replacement map"
, Option "d" ["dim"]
(ReqArg (\arg opt -> do return opt { optDim = (read (arg) :: Int) })
"DIMENSION"
)
"Dimension"
, Option "n" ["interpretation"]
(ReqArg (\arg opt -> do return opt { optInterpretation = arg })
"INTERPRETATION"
)
"Interpretation type [Linear]"
, Option "p" ["property"]
(ReqArg (\arg opt -> do return opt { optPropName = arg
,optPropInput = readFile arg})
"FILE"
)
"Properties to test"
, Option "v" ["version"]
(NoArg (\_ -> do hPutStrLn stderr "muTerm, version 5.0"
exitWith ExitSuccess))
"Print version"
, Option "t" ["timeout"]
(ReqArg (\arg opt -> do to <- readArg "timeout" arg
scheduleAlarm to
installHandler sigALRM
(Catch (putStrLn "MAYBE"
>> exitImmediately (ExitFailure 2))
) Nothing
return opt {optTimeout = Just to})
"SECONDS"
)
"Specify a timeout in seconds"
]
```


Terminal - patolin@patolin-vb: ~/Documentos/TFM
Archivo Editar Ver Terminal Pestañas Ayuda

```
patolin@patolin-vb:~/Documentos/TFM$ ./Main
use -i option to set input

Main
  -h                  --help                            Show usage info
  -i FILE             --input=FILE                      Input file
                      --html                            Output in HTML format
                      --xml                             Output in XML format
  -c                  --canonical                       Return the input system adding the canonical replacement map
                      --pato                            Texto de prueba
  -d DIMENSION        --dim=DIMENSION                   Dimension
  -n INTERPRETATION   --interpretation=INTERPRETATION   Interpretation type [Linear]
  -p FILE             --property=FILE                   Properties to test
  -v                  --version                         Print version
  -t SECONDS          --timeout=SECONDS                 Specify a timeout in seconds

patolin@patolin-vb:~/Documentos/TFM$
```

Figure 5.1: Command Line Arguments

With this modification of the original code the tool can handle the required extra parameters. Invoking the tool without any parameter (or including the help command) displays a brief description of available parameters and settings as shown in Figure 5.1.

5.3 New Modules

5.3.1 Goal Parser

The property to be tested must be uploaded from a separate file. In this case, a *.prop* file loaded using the −p command line argument.

Example 25

For our running example, the goal is a file containing the following sentences:

```
wwv05(U:User) ->* submit(U:User) => ~(U:EventualUser)
wwv05(U:User) ->* submit(U:User) => U:RegUser
```

Two new modules were created to parse such goals:

1. The Parser.PROP.Parser module contains the function parsePROP, which has the type:

```
parsePROP
:: Framework.Problem.Types.MProblem t
-> InferenceRules.ToConstraint.Interp
-> String
-> Either
ParseError
(Constraint.Constraints2.CCode
(Constraint.Polynomials3.UV String)
(Constraint.Polynomials3.Poly (Constraint.Polynomials3.UV String)))
```

In fact, this function takes as parameters:

(a) the TRS obtained from parsing the MAUDE program,

(b) the set of properties to be demonstrated as a String, and

(c) the interpretation type, for function symbols.

Note that this is a partial function; for this reason the OSTRS type is not included in the function signature. This function calls the `propParserTransform` function located in the `propParserTransform` module.

2. The `propParserTransform` function, has the following type:

```
propParserTransform
:: MProblem t
-> InferenceRules.ToConstraint.Interp
-> String
-> Either ParseError (CCode (UV String) (Poly (UV String)))
```

This function takes the parsed OS-TRS, the interpretation type, and the string with the contents of the .prop file, and returns either an error from the parser, or a `(CCode (UV String) (Poly (UV String)))` polynomial that will be the constraint representation of the program.

To parse the goal, we use, the *Parsec* package. Parsec is a monadic parser for Haskell, it's very simple to use[2]. In order to parse our goal formula, the first thing to do is declaring the reserved names and operator names. For that, we define a lexer function for the reserved function names `true`, `false` and for the logical operators `->`, `->*`, `=>`, `:`, `~`, `\/`, `/\`.

```
lexer = P.makeTokenParser fun
where fun = haskellStyle {
reservedNames   = ["true", "false"]
, reservedOpNames = ["=>", "\\/", "/\\",
"~", "->*", "->",
":", ","]
}
```

Also, we need to define how the reserved operators use their parameters when building expressions. An extract of the complete code (just to see an example of use) will be like the following:

[2]Full documentation for *Parsec* can be found at https://hackage.haskell.org/package/parsec/

```
opSpec :: Parser Spec
opSpec = buildExpressionParser table apSpec
where table = [[prefix "~" FNot],
binary "->" (:->) AssocLeft,
binary "->*" (:->*) AssocLeft,
binary "::" (:::) AssocLeft
],
[binary "=>" (:=>) AssocLeft],
[binary "\\/" (:\/) AssocLeft,
binary "/\\" (:/\) AssocLeft ]
]
binary s op assoc = Infix (do {reservedOp s; return op}) assoc
prefix a b = Prefix (do{reservedOp a; return b})
```

After parsing the goal formula (which has a `Formula a b` type) a transformation is still needed to convert this formula into data with a MU-TERM defined type. In order to achieve this, we have functions `transformFormula` and `transformTermino`, which take the goal parsed in the previous stage, and transform it to a value of type `Term (IdFOS (IdFAvC IdFTRS)) (IdVOS IdVTRS)`.

An excerpt of the code is shown to display the structure of those functions.

```
transformFormula _ (FTrue)       = FTrue
transformFormula _ (FFalse)      = FFalse
transformFormula trs (FNot a)    = FNot $ transformFormula trs a
transformFormula trs (a :/\ b)   =
transformFormula trs a :/\ transformFormula trs b
transformFormula trs (a :\/ b)   =
transformFormula trs a :\/ transformFormula trs b
transformFormula trs (a :=> b)   =
transformFormula trs a :=> transformFormula trs b
transformFormula trs (a :-> b)   =
Rel $ transformTermino trs a ::-> transformTermino trs b
transformFormula trs (a :->* b)  =
Rel $ transformTermino trs a ::->* transformTermino trs b
transformFormula trs (a ::: b)   =
Rel $ transformVariable trs a :-: transformSort trs b
transformFormula trs (Termino a) =
error ("Se espera una formula pero se obtiene el termino: "++show(a))

transformFormula' (FNot t)       = FNot $ transformFormula' t
transformFormula' (Termino t)    = transformTermino'' t

transformTermino trs (Termino t) = transformTermino' trs t

transformTermino' :: MProblem t
-> Termi
-> Term (IdFOS (IdFAvC IdFTRS)) (IdVOS IdVTRS)
transformTermino' trs (Fun n terms) = case (symbol) of
```

```
Nothing -> error ("Simbolo n no encontrado en signatura: " ++ show n)
Just f -> F f (map (transformTermino' trs) terms)
```

After this process, all the information (the original TRS, and the parsed goal) are stored in the *trs* variable, which is used in the next step, to generate the required inference rules.

5.3.2 Generating convex domains

The next step is the generation of the (parametric) *matrices* C_s and *vectors* $\vec{b}_s$ for each sort s defined in the specification. We proceed as follows:

1. All sorts declared in the program are listed.
2. A parametric convex domain matrix C_s is associated to each sort s, and saved in a list, for later access. Similarly for the vector component $\vec{b}_s$ of the domain.

The functions implementing these tasks are part of `InferenceRules.Inference` and `InferenceRules.GenMatrix` modules.

First, the list of sorts from the original program, is obtained by means of two actions:

1. Obtain the list of sorts and subsorts using the `extractSorts` function from the `Inference.Inference` module.
2. Generate fresh variables for each sort listed, using the `varsFromSort` function from the `Inference.GenMatrix` module. These variables are used as *markers* to be replaced in the constraints defining the values belonging to the convex domains by variables of the corresponding sort when other formulae are treated (see Section 5.3.4 below, for instance).

The functions required for this task are defined as follow:

```
extractSorts trs = superSort where
getMMaudeProblem (MMaudeProblem a) = getR a
ostrs = getMMaudeProblem trs
listaSorts = getOSSorts ostrs
listaPrec = getOSPrec ostrs
getTopSorts a = Prelude.filter (\node -> pre a node == []) (nodes a)
listaSuperSorts = getTopSorts listaPrec
superSort = elems listaSorts

varsFromSort [] _ = return []
varsFromSort (x:xs) v = do
a1 <- getInt
cola <- varsFromSort xs v
let var1 = newvar (setOutSort x v) a1
let salida = (UV ("xs_"++(show a1)) [] (Just x)):cola
return salida
```

With a list of fresh variables a_i and b_i for $i \in \mathbb{N}$ (automatically generated using `MonadState`) the tool generates the convex matrix for each new variable (or sort in this specific case, because the new variable is just a container for the sort). To perform this task, we use the following functions:

```
genMatrixVar :: MonadState Int m
=> Int
-> [UV [Char]]
-> m [(PCode (UV [Char]), (CCode ((UV [Char])) (PCode (UV [Char]))))]
genMatrixVar dim var = do
salida <- sequence ( map (funcMatrices' dim) var)
return salida

funcMatrices' :: MonadState Int m
=> Int
-> UV [Char]
-> m (PCode (UV [Char]), (CCode (UV [Char]) (PCode (UV [Char]))))
funcMatrices' dim variable = do
let varX = var variable
let listaIDs = [ i | i <- [1..dim]]
formula <- sequence( map (generaRestriccion varX variable) listaIDs)
let formulaFlat = foldl1 (and) formula
let salida = (varX, formulaFlat)
return salida
```

The result of the previous process is a list [(`var`, `convex_matrix_constraints`)] of tuples where

1. the first component of the tuple (*var*) contains the name and sort of the variable $\vec{x}$ which is used to define the values in the convex domain as the set $D(\mathsf{C}_s, \vec{b}_s)$ of solutions of the linear inequality $\mathsf{C}_s\vec{x} \geq \vec{b}_s$ (see Definition 12) and
2. the second component (*convex_matrix_constraints*) contains the constraints that correspond to the use of the convex domain to represent values of the sort s in the first element of the tuple. The constraints are obtained according to the selected dimensions (number of rows, m_s, and columns, n_s) of C_s. Note that $\vec{x} \in \mathbb{R}^{n_s}$

Example 26 (Convex domains for the running example.)
As for our running example, we let $m_s = m = 2$ and $n_s = n = 1$ for all sorts s to obtain matrices C and vectors $\vec{b}$ of the generic shape

$$\mathsf{C} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

where a_1, a_2, b_1 and b_2 are *parameters* that are different for each sort s. Note that, since $n = 1$, variables $\vec{x}$ ranging on $D(\mathsf{C}, \vec{b})$ are 1-dimensional vectors, i.e., $\vec{x}$ actually ranges on a subset of numbers. Since only one column is used for the matrix, we can just show them as vectors, thus saving space: $(a_1, a_2)^T$ and $(b_1, b_2)^T$.

For the sake of readability, we shortened the sort names associated to of each *semantic* variable. The correspondence with such shortened sort names is:

`User`	`U`
`RegUser`	`RU`
`EventualUser`	`EU`
`WebPage`	`WP`
`SecureWebPage`	`SWP`

Now we show the outcome of our generation procedure for each of the five sorts in the program:

```
[
(xs_1:EU,
(((a_6:EU * xs_1:EU) >= b_7:EU)
/\ ((a_8:EU * xs_1:EU) >= b_9:EU))),
(xs_2:RU,
(((a_10:RU * xs_2:RU) >= b_11:RU)
/\ ((a_12:RU * xs_2:RU) >= b_13:RU))),
(xs_3:U,
(((a_14:U * xs_3:U) >= b_15:U)
/\ ((a_16:U * xs_3:U) >= b_17:U))),
(xs_4:WP,
(((a_18:WP * xs_4:WP) >= b_19:WP)
/\ ((a_20:WP * xs_4:WP) >= b_21:WP))),
(xs_5:SWP,
(((a_22:SWP * xs_5:SWP) >= b_23:SWP)
/\ ((a_24:SWP * xs_5:SWP) >= b_25:SWP)))
]
```

For instance, according to the information in this list of tuples, the matrix of parameters that corresponds to matrix $\mathsf{C}_{\mathtt{EU}}$ for the convex domain of sort `EventualUser` is $(\mathtt{a\_6}, \mathtt{a\_8})^T$. Similarly, $\vec{b}_{\mathtt{EU}}$ is $(\mathtt{b\_7}, \mathtt{b\_9})^T$. The concrete numbers accompanying the identifiers `a` and `b` (that we use for identifying the coefficients of matrices C and vectors $\vec{b}$, respectively) are given automatically by `MonadState`.

Furthermore, the constraints related to the subsort relationships are generated as in (3.21).

Remark 27 All functions listed above (and other that will appear later) use the State Monad, because they use an internal counter to assign a unique id (an *integer id*) to each fresh variable or function. ■

5.3.3 Generating Polynomial Interpretations

Also before generating the inference rules for our theory, we need to define how the *Polynomial Interpretations* are generated for each function symbol. This kind of interpretation will be necessary before we generate all the constraints for the theory we are modeling. To do this, we proceed as follows:

- Identify the function symbol in our OS-TRS.
- For each function symbol, generate a polynomial interpretation, in order to replace the function with its own interpretation in the rule.

For this step in the process, we use the `Constraint.Interpretations2` module from MU-TERM, which includes the function `polyIntSigSort`. This function handles the interpretation of a function symbol from our OS-TRS, in our case, we use *linear* polinomyal interpretations only. Therefore, this function invokes directly to `LinearPolyIntSigSort`, which is an adaptation of the `LinearPolyIntSig` function, modified to work with OS-TRS. The function creates a polynomial interpretation for a function symbol, and requires the following parameters:

- `Dimension`: The dimension of the coefficients (in our case it is settled to 1)
- `Coefficients`: The type of coefficients used for the interpretation (In our case, fixed to *integer* numbers)
- `fs`: The function symbol to be interpreted.

```
-- | create a linear polynomial interpretation
linearPolyIntSigSort :: (Ord idF, Pretty idF, HasName idF,
HasFSort idF, Polynomial a (UV String), HasArity idF)
=> Dimension
-> Coefficients
-> Set idF
-> Map idF ([UV String], a)
linearPolyIntSigSort  dim coeffs functions =
M.fromList [(f,linearPolyIntSort
dim coeffs f (getFArity f)) |
f <- S.elems functions]
```

We illustrate the execution of this interpretation function, with the following example:

Example 28

The function symbol:

    login(U)

where the function symbol `login` has the sort `Website`, and the variable `U` has the sort `User`, is interpreted as:

```
[login](v1:User) = f0_0:WebPage + (f0_1:WebPage * v1:User)
```

where `f0_0` are `f0_1` are automatically generated *parameters*, i.e., unknown coefficients of the polynomial interpretation which will be settled by the constraint solving process described in the last sections of this chapter.

For each function symbol, we generate the algebraicity constraints as the ones presented in (3.29)-(3.35).

5.3.4 Specialization of the input specification using the inference rules

By using the list of tuples that describe the different domains by means of appropriate linear constraints, we can proceed to generate the logic formulae, by specializing the rules in Figure 1.1.

Reflexive Inference Rules

The Reflexive Inference Rule

$$\frac{}{t \to^* t}$$

means that for each connected component in [s] each term t can be rewriten in one or more steps to the same term t. In fact, the inference rule can be thought of as a variable of superior sort [s] that can be bound to any term of sort in [s]. For this, the module `InferenceRules.Inference` includes two functions to generate this rules.

```
infRefl trs matriz interParams symbols = do
  let listaVars = (elems . getTRSVariables $ trs)
  reglas <- sequence
    ( map (genRef symbols interParams matriz) listaVars)
  return reglas

genRef symbols interParams matriz v = do
  a1 <- getInt
  let var1 = newvar' v a1
  let reglaInferencia = (Rel ( (var1) ::->* (var1) ) )
  let resultado = generaConstraints
  reglaInferencia matriz symbols interParams
  return resultado
```

The function `infRefl` takes the following parameters:

- the TRS parsed from the Maude program.
- the list of tuples describing the convex domains for each sort that was created in the previous step (which includes the constraints for all sorts of the TRS).

- the interpretation parameters (including the matrix dimension and the interpretation type, parsed from the command line arguments).
- and the list of function symbols extracted from the TRS.

Remark 29 The aforementioned parameters are *almost the same for all functions* that generate the distinct inference rules explained in the remainder sections. ■

Function `infRefl` applies function `genRef` to all variables extracted from the TRS. The function `genRef` takes a variable and creates a fresh variable *using the State Monad to generate a new variable Id* with the function `newvar'`, wich creates the fresh variable, taking the new Id, and the sort from the actual one.

```
newvar' :: (HasName idV) => idV -> Int -> Term idF idV
newvar' v a = V ((setName (Nothing) . setId (a+1) $ v))
```

This new variable is used to create a new rule `Rel (v ->* v)`. This rule is passed to the `generaConstraints` function, that takes the generated inference rule, and adds the corresponding constraints according to the sort of the variables used. The result of this functions is shown in the following example:

Example 30 (Specialization of the reflexivity rule in the running example)

With regard to our running example, we have to specialize the reflexivity rule for the two connected components in the sort hierarchy: [`User`] and [`WebPage`]. The list of obtained formulae (implications) is:

```
[
(  (((a_14:U * v27:U) >= b_15:U)
/\  ((a_16:U * v27:U) >= b_17:U))
=>
     (v27:U >= v27:U)
),
(  (((a_18:WP * v28:WP) >= b_19:WP)
/\  ((a_20:WP * v28:WP) >= b_21:WP))
=>
     (v28:WP >= v28:WP)
)
]
```

Note that, as explained above, the variables `xs_3` (for sort `User`) and `xs_4` (for sort `WebPage`) which are obtained in Example 26 to represent the domains `User` and `WebPage`, respectively, are used as markers to be *replaced* here by variables `v27` and `v28` in the specialization of the corresponding inference rules to obtain the two implications.

Transitive Inference Rules

The generation of the rules corresponding to the generic Transitive Inference Rule

$$\frac{t \to t' \qquad t' \to^* u}{t \to^* u}$$

is similar what we do for the Reflexive Rule. In this case, though, we need to create *three* fresh variables per connected component [s] in the sort hierarchy.

```
infTran trs matriz interParams symbols = do
  let listaVars = (elems . getTRSVariables $ trs)
  reglas <- sequence ( map (genTra symbols interParams matriz) listaVars)
  return reglas

genTra symbols interParams matriz v = do
  v1 <- getInt
  v2 <- getInt
  v3 <- getInt
  let reglaInferencia = ((Rel ( (newvar' v v1) ::->  (newvar' v v2) )
    :/\  Rel ( (newvar' v v2) ::->* (newvar' v v3)) )
    :=> (Rel ( (newvar' v v1) ::->* (newvar' v v3))) )
  let resultado = generaConstraints
  reglaInferencia matriz symbols interParams
  return resultado
```

With these functions, we create rules of the form

$$\texttt{Rel(var1 -> var2)} \wedge \texttt{Rel(var2 ->* var3)} \Rightarrow \texttt{Rel(var1 ->* var3)}$$

Example 31 (Specialization of the transitivity rule in the running example)

After introducing the information about convex domains for the specific sorts of the variables (`User` and `WebPage`), the outcome of this step for our example is:

```
[
(((((a_14:User * v31:User) >= b_15:User)
/\  ((a_16:User * v31:User) >= b_17:User))
/\ (((a_14:User * v32:User) >= b_15:User)
/\  ((a_16:User * v32:User) >= b_17:User)))
/\ (((a_14:User * v33:User) >= b_15:User)
/\  ((a_16:User * v33:User) >= b_17:User))))
/\ (((v31:User > v32:User)
/\   (v32:User >= v33:User))
=>
     (v31:User >= v33:User)
,
(((((a_18:WebPage * v34:WebPage) >= b_19:WebPage)
/\  ((a_20:WebPage * v34:WebPage) >= b_21:WebPage))
/\ (((a_18:WebPage * v35:WebPage) >= b_19:WebPage)
/\  ((a_20:WebPage * v35:WebPage) >= b_21:WebPage)))
/\ (((a_18:WebPage * v36:WebPage) >= b_19:WebPage)
/\  ((a_20:WebPage * v36:WebPage) >= b_21:WebPage))))
/\ (((v34:WebPage > v35:WebPage)
/\   (v35:WebPage >= v36:WebPage))
=>
     (v34:WebPage >= v36:WebPage)
)
]
```

The coefficients a from the constraints, correspond to the restriction of taking values in the domain of the respective sort, as in the reflexive inference rules generated in previous steps.

Congruence Inference Rule

For the Congruence Inference Rule, we have to generate inference rules from the generic rule

$$\frac{t_i \to t_i'}{f(t_1, \ldots, t_i, \ldots, t_k) \to f(t_1, \ldots, t_i', \ldots, t_k)}$$

for each function symbol f in the signature (with rank $f : s_1 \cdots s_k \to s$), and each argument i of the symbol (i.e., $i \in \{1, \ldots, k\}$) using the appropriate relations for the sort s_i of the i-th argument and for the outocome sort s of symbol f. Of course, $t_1, \ldots, t_k$ are variables ranging over the corresponding convex domains $D(\mathsf{C}_{s_1}, \vec{b}_{s_1}), \ldots, D(\mathsf{C}_{s_k}, \vec{b}_{s_k})$ and t_i' ranges on $D(\mathsf{C}_{s_i}, \vec{b}_{s_i})$.

The functions that handle this inference rule are:

```
filterEmpty [] = []
filterEmpty (x:xs)
  | length(x)>0  = x : filterEmpty xs
  | otherwise    = filterEmpty xs

genVarsList :: (HasName (IdVOS IdVTRS), Num t, MonadState Int m, Enum t)
    => (IdVOS IdVTRS)
    -> t
    -> m [Term (IdFOS (IdFAvC IdFTRS)) (IdVOS IdVTRS)]
genVarsList v arity = sequence ([ newvar'' v | ar <- [0..(arity-1)]])

newvar'' :: (MonadState Int m, HasName (IdVOS IdVTRS)) => (IdVOS IdVTRS)
    -> m (Term (IdFOS (IdFAvC IdFTRS)) (IdVOS IdVTRS))
  newvar'' v = do
  idV1 <- getInt
  let v1 = newvar' v idV1
  return v1

infCong trs matriz interParams symbols = do
  let listaFuncs = (elems $ getTRSSymbols trs)
  let v = findMax $ getTRSVariables trs
  reglas' <- sequence ( map (genCon symbols interParams matriz v) listaFuncs)
  let reglas = filterEmpty reglas'
  return  (concat reglas)

genCon symbols interParams matriz v funcion= do
  let inSorts = getInSorts . head . getFSorts $ funcion
  let fArity = getFArity funcion
  listaVarsF <- varsFromSort'' inSorts v
  reglas <- sequence ( [genConFormula funcion listaVarsF v pos matriz
  symbols interParams| pos <- [0..(fArity-1)] ] )
  return reglas

genConFormula f listaVars v pos matriz symbols interParams= do
  idV1 <- getInt
  idV2 <- getInt
  let (x,_:ys) = splitAt pos listaVars
  let removeVfromVar (V a) = a
  let v1 = newvar' (removeVfromVar(listaVars!!pos)) idV1
  let v2 = newvar' (removeVfromVar(listaVars!!pos)) idV2
  let v0a = x ++ [v1] ++ ys
  let v0b = x ++ [v2] ++ ys
  let f1 = newfun f v0a
  let f2 = newfun f v0b
  let reglaInferencia = ( (Rel (v1 ::-> v2)) :=>  (Rel (f1 ::-> f2)))
  let resultado = generaConstraints reglaInferencia matriz symbols interParams
  return resultado
```

First, in the function `infCong` we extract the function symbols from the original TRS, and for each symbol, make a call to the `genCon` function, on order to get the *arity* of that function, generate a new list of variables based on the function arity and sorts of the input variables, and using the function `genConFormula`, replace each variable inside the new function symbols (named `f1` and `f2` in our code) in order to complete the inference rule. After we include the convex matrices and transform to a polynomial interpretation of the functions, we obtain this result from our running example:

Example 32 (Specialization of the congruence rule in the running example)

```
[
     ((v38:User > v39:User)
/\ ((((a_14:User * v38:User) >= b_15:User)
/\   ((a_16:User * v38:User) >= b_17:User))
/\  (((a_14:User * v39:User) >= b_15:User)
/\   ((a_16:User * v39:User) >= b_17:User))))
=>
     ((f0_0:WebPage + (f0_1:WebPage * v38:User))
    > (f0_0:WebPage + (f0_1:WebPage * v39:User))),

     ((v41:User > v42:User)
/\ ((((a_14:User * v41:User) >= b_15:User)
/\   ((a_16:User * v41:User) >= b_17:User))
/\  (((a_14:User * v42:User) >= b_15:User)
/\   ((a_16:User * v42:User) >= b_17:User))))
=>
     ((f1_0:WebPage + (f1_1:WebPage * v41:User))
    > (f1_0:WebPage + (f1_1:WebPage * v42:User))),

((v44:User > v45:User)
/\ ((((a_14:User * v44:User) >= b_15:User)
/\   ((a_16:User * v44:User) >= b_17:User))
/\  (((a_14:User * v45:User) >= b_15:User)
/\   ((a_16:User * v45:User) >= b_17:User))))
=>
     ((f2_0:WebPage + (f2_1:WebPage * v44:User))
    > (f2_0:WebPage + (f2_1:WebPage * v45:User))),

     ((v47:User > v48:User)
/\ ((((a_14:User * v47:User) >= b_15:User)
/\   ((a_16:User * v47:User) >= b_17:User))
/\  (((a_14:User * v48:User) >= b_15:User)
/\   ((a_16:User * v48:User) >= b_17:User))))
=>
     ((f3_0:SecureWebPage + (f3_1:SecureWebPage * v47:User))
    > (f3_0:SecureWebPage + (f3_1:SecureWebPage * v48:User))),

     ((v50:User > v51:User)
/\ ((((a_14:User * v50:User) >= b_15:User)
/\   ((a_16:User * v50:User) >= b_17:User))
/\  (((a_14:User * v51:User) >= b_15:User)
/\   ((a_16:User * v51:User) >= b_17:User))))
=>
     ((f4_0:WebPage + (f4_1:WebPage * v50:User))
    > (f4_0:WebPage + (f4_1:WebPage * v51:User))),

     ((v53:User > v54:User)
/\ ((((a_14:User * v53:User) >= b_15:User)
/\   ((a_16:User * v53:User) >= b_17:User))
/\  (((a_14:User * v54:User) >= b_15:User)
/\   ((a_16:User * v54:User) >= b_17:User))))
=>
     ((f5_0:WebPage + (f5_1:WebPage * v53:User))
    > (f5_0:WebPage + (f5_1:WebPage * v54:User))),
     ((v56:RegUser > v57:RegUser)
/\ ((((a_10:RegUser * v56:RegUser) >= b_11:RegUser)
/\   ((a_12:RegUser * v56:RegUser) >= b_13:RegUser))
/\  (((a_10:RegUser * v57:RegUser) >= b_11:RegUser)
/\   ((a_12:RegUser * v57:RegUser) >= b_13:RegUser))))
=>
     ((f6_0:SecureWebPage + (f6_1:SecureWebPage * v56:RegUser))
    > (f6_0:SecureWebPage + (f6_1:SecureWebPage * v57:RegUser)))
]
```

Replacement Inference Rules

The specialization of the Replacement Inference Rules

$$\frac{}{\ell \to r}$$

for each rule $\ell \to r$ in the program, uses two functions to generate a formula with format `Rel(`ℓ `->` r`)`:

```
infRepl trs matriz interParams symbols = do
  let listaFuncs = (elems $ getTRSSymbols trs)
  let v = findMax $ getTRSVariables trs
  let listaReglas = elems $ getTRSRules trs
  reglas <- sequence
    ( map (genRep symbols interParams matriz) listaReglas)
  return reglas

genRep symbols interParams matriz regla = do
  let lhs = T.lhs(regla)
  let rhs = T.rhs(regla)
  let reglaInferencia = Rel (lhs ::-> rhs)
  let resultado = generaConstraints reglaInferencia matriz symbols interParams
  return resultado
```

With this, and the remark that the rule `Rel`(`lhs` $::->$ `rhs`) and its terms are transformed into their polynomial interpretation, using the `toConstraint` function, the result for the replacement inference rules is:

Example 33 (Specialization of the replacement rule in the running example)

```
[
    (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User)))
=>
```

```
    (((f0_0:WebPage + (f0_1:WebPage * v1:User))
>     (f3_0:SecureWebPage + (f3_1:SecureWebPage * v1:User))),

    (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User)))
=>
    (((f1_0:WebPage + (f1_1:WebPage * v1:User))
>     (f0_0:WebPage + (f0_1:WebPage * v1:User))),

    (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User)))
=>
    (((f1_0:WebPage + (f1_1:WebPage * v1:User))
>     (f5_0:WebPage + (f5_1:WebPage * v1:User))),

    (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User)))
=>
    (((f2_0:WebPage + (f2_1:WebPage * v1:User))
>     (f1_0:WebPage + (f1_1:WebPage * v1:User))),

    (((a_10:RegUser * v0:RegUser) >= b_11:RegUser)
/\   ((a_12:RegUser * v0:RegUser) >= b_13:RegUser)))
=>
    (((f3_0:SecureWebPage + (f3_1:SecureWebPage * v0:RegUser))
>    (f6_0:SecureWebPage + (f6_1:SecureWebPage * v0:RegUser))),

   ((((a_10:RegUser * v0:RegUser) >= b_11:RegUser)
/\   ((a_12:RegUser * v0:RegUser) >= b_13:RegUser))
/\  (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User))))
=>
    (((f3_0:SecureWebPage + (f3_1:SecureWebPage * v1:User))
>     (f6_0:SecureWebPage + (f6_1:SecureWebPage * v0:RegUser))),

/\  (((a_14:User * v1:User) >= b_15:User)
/\   ((a_16:User * v1:User) >= b_17:User)))
=>  (((f4_0:WebPage + (f4_1:WebPage * v1:User))
>     (f2_0:WebPage + (f2_1:WebPage * v1:User)))
]
```

Goal Constraint Generation

For the goal to be verified, we execute the same process, inserting the information about the domain for variables in the formula by adding the constraints for the sorts of the variables used in that formula.

Example 34

For the last goal in Example 25, i.e.,

```
wwv05(U:User) ->* submit(U:User) => U:RegUser
```

the application of the constraint generation functions, yields the following restriction:

```
[
((
(((a_14:U * v0:U) >= b_15:U)
/\  ((a_16:U * v0:U) >= b_17:U))
/\   (f4_0:WP + (f4_1:WP * v0:U)) >= (f0_0:WP + (f0_1:WP * v0:U)) )
=>  ((a_10:RU * v0:U >= b_11:RU) /\ (a_12:RU * v0:U >= b_13:RU))
)
]
```

where `f4_0` and `f4_1` are the parameters interpreting symbol `wwv05` as a *linear* polynomial `f4_1`$x +$`f4_0`, where x ranges over values of sort `User`, according to the rank of `wwv05` in program `WWV05-WEBSITE`. Similarly, `f0_0` and `f0_1` are the parameters interpreting symbol `submit` as a *linear* polynomial `f0_1`$x +$`f0_0`, where x ranges over values of sort `User` as well.

Finally, collect all constraints by means of Haskell's predefined `and` function, so that all constraints are handled as a single conjunction of (conditional) linear constraints. This is stored in a single variable to be used in the next steps of the tool.

```
let restricciones = and algSig' (and subsRel' infRules')
let infRules' = and refl' (and tran' (and cong' repl'))
```

5.3.5 Final Transformations and Solver Execution

The outcome of the previous process is a conjunction $\bigwedge_{i=1}^{n} P_i \Rightarrow Q_i$ where (in general), P_i and Q_i are *conjunctions* of expressions $s_i \bowtie t_i$ for polynomials s_i and t_i and where $\bowtie$ is a relation $\geq$, $>$ or $=$. All semantic variables in this formula are universally quantified over the domains of the corresponding sorts. The parameters introduced during the generation process are intended to be *existentially quantified* and will be given appropriate values through a *constraint solving* process using a constraint solving tool.

Before being able to use the *solver*, we need to apply some further transformations. These transformations are mostly available from modules `Constraint.Constraints2`, `Constraint.Polynomials3` and `Constraint.Interpretations2`) of MU-TERM. We briefly explain their functionallity as follows.

In the AGES code, there are six steps required after the constraint generation, that are mandatory before we send the constraints to the solver tool.

- Transform the logic formulae into arithmetic constraints.

- Remove conditional implications and remove universally quantified variables using Farkas's theorem.
- Simplify constraints.
- Add a constraint for the *delta* variable, which is used to obtain a discrete domain $>$.

The code that executes those tasks is the following:

```
let p0vars = S.filter
(Prelude.null . uvinfo) . getFreeVarsInCCode $ restricciones
p1 <- removeConds (restricciones)
let p3 = simplifyPols p2
let p5 = and (gr delta zero) p3
```

1. First of all, the following command:

   ```
   let p0vars = S.filter (Prelude.null . uvinfo)
   . getFreeVarsInCCode $ restricciones}
   ```

   takes the constraint generated before, and extracts the free (semantic) variables. This process is made by analizing the data structure of each variable in the constraint, and the ones which their `uvinfo` value is `null`, are included in a list.

   Example 35

   For our running example, the free variables obtained from the constraint are:

   ```
   v0:RU    v1:U    v27:RU   v28:U v29:RU   v30:RU
   v31:RU   v32:U   v33:U    v34:U v35:U    v36:U
   v37:U    v38:U   v39:U    v40:U v41:U    v42:U
   v43:U    v44:U   v45:U    v46:U v47:U    v48:U
   ```

2. The command

   ```
   p1 <- removeConds (restricciones)
   ```

   takes the constraints defined with $>$ and transforms them into a $\geq$-based constraint by interpreting $>$ as the following well-founded ordering $>_\delta$ over the integers for a given positive number δ: $x >_\delta y$ if and only if $x \geq y + \delta$ [12]. This step is necessary because to work with Farkas's theorem, all the constraints must have inestrict inequalities.

   Example 36

   The constraint

```
(a_10:RU * v27:RU) > b_11:RU
```

is transformed into:

```
(a_10:RU * v27:RU) >= b_11:RU + delta
```

At the end of the transformation process, we will add a new constraint `delta > 0` to guarantee the correctness of the approach. The important point is that `delta` is existentially quantified in the final formula and can be handled by the constraint solving tool.

We can use now the techniques discussed in [11] to transform conditional polynomial constraints into constraint solving problems. In particular the Affine form of Farkas' Lemma considered in [11, Section 5.1], can be used to deal with linear conditional constraints like (3.29)-(3.47) that consist of *implications* $\bigwedge_{j=1}^{p_i} e_{ij} \geq d_{ij} \Rightarrow e_i \geq d_i$, where for all $i \in \{1, \ldots, k\}$, $p_i > 0$ and for all j, $1 \leq j \leq p_i$, e_{ij} and e_i are linear expressions and $d_{ij}, d_i \in \mathbb{R}$. We say that these implications are in *affine form*. In general, given $\vec{c} \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$, the affine form of Farkas' Lemma can be used to check whether a constraint $\vec{c}^T\vec{x} \geq \beta$ holds whenever $\vec{x}$ ranges on the set S of solutions $\vec{x} \in \mathbb{R}^n$ of a linear system $A\vec{x} \geq \vec{b}$ of k inequalities, i.e., A is a matrix of k rows and n columns and $\vec{b} \in \mathbb{R}^k$. According to Farkas' Lemma, we have to *find* a vector $\vec{\lambda}$ of k non-negative numbers $\vec{\lambda} \in \mathbb{R}_0^k$ such that $\vec{c} = A^T\vec{\lambda}$ and $\vec{\lambda}^T\vec{b} \geq \beta$.

Example 37

We can write (3.29) to fit the *affine* form above as follows:

$$C_1^u x \geq b_1^{\mathtt{U}} \wedge C_2^u x \geq b_2^{\mathtt{U}} \quad \Rightarrow \quad C_1^w l_1 x \geq b_1^{\mathtt{WP}} - C_1^w l_0 \tag{5.1}$$
$$C_1^u x \geq b_1^{\mathtt{U}} \wedge C_2^u x \geq b_2^{\mathtt{U}} \quad \Rightarrow \quad C_2^w l_1 x \geq b_2^{\mathtt{WP}} - C_2^w l_0 \tag{5.2}$$

Note that we obtain a logically equivalent set of two implications due to the conjunction of two affine inequalities in the consequent of (3.29). Now, we apply Farkas' lemma to each of them. Both (5.1) and (5.2) have the same associated matrix A, which is actually a *vector* $(C_1^u, C_2^u)^T$. Similarly, we have the same vector $\vec{b} = (b_1^{\mathtt{U}}, b_2^{\mathtt{U}})^T$. For (5.1) $\vec{c}$ is actually a one-dimensional vector $C_1^w l_1$ and $\beta = b_1^{\mathtt{WP}} - C_1^w l_0$ for both of them. For (5.2) $\vec{c}$ is $C_2^w l_1$ and $\beta = b_2^{\mathtt{WP}} - C_2^w l_0$.

For (5.1), the application of Farkas' Lemma seeks a vector $\vec{\lambda} = (\lambda_1, \lambda_2)^T$ with $\lambda_1, \lambda_2 \geq 0$ that satisfies the following two (in)equations:

$$C_1^w l_1 = C_1^u \lambda_1 + C_2^u \lambda_2 \qquad \lambda_1 b_1^{\mathtt{U}} + \lambda_2 b_2^{\mathtt{U}} \geq b_1^{\mathtt{WP}} - C_1^w l_0$$

for some values of the parameters $C_1^u, C_2^u, C_1^w, b_1^{\mathtt{U}}, b_2^{\mathtt{U}}, l_0, l_1$. For (5.2), we seek some $\vec{\lambda}' = (\lambda_1', \lambda_2')^T$ with $\lambda_1', \lambda_2' \geq 0$ that satisfies:

$$C_2^w l_1 = C_1^u \lambda_1' + C_2^u \lambda_2' \qquad \lambda_1' b_1^{\mathtt{U}} + \lambda_2' b_2^{\mathtt{U}} \geq b_2^{\mathtt{WP}} - C_2^w l_0$$

for some values of the parameters $C_1^u, C_2^u, C_2^w, b_1^{\mathtt{U}}, b_2^{\mathtt{U}}, l_0, l_1$. Note, however, that the values associated to $C_1^u, C_2^u, C_1^w, C_2^w, b_1^{\mathtt{U}}, b_2^{\mathtt{U}}, l_0, l_1$ should be *the same* for both (5.1) and (5.2) as they represent ingredients defining *the same* semantic structure. Actually, this observation is valid for *all* parameters occurring in (3.21)-(3.51). For this reason, although each implication processed using Farkas' Lemma can use a *different* vector $\vec{\lambda}$, we have to *solve a single set of inequations corresponding to a single solution which produces a single model that makes all sentences valid.*

3. Now we proceed to further simplify the constraints. The process is just an algebraic simplification that the following example illustrates.

   Example 38 (Simplification process)

   In the equation:

   ```
   v27:RU >= v27:RU
   ```

   we *move* the term in the right side to the left side, so the inequation becomes:

   ```
   v27:RU - v27:RU >= 0
   ```

   And after an algebraic simplification we obtain:

   ```
   0 >= 0
   ```

   Which is true. We can therefore *remove* this constraint from any conjunction containing it.

   The function `simplifyPols` found in the `Constraint.Constraints2` module, applies the simplification to all generated constraints. This function takes as input a `(CCode a (PCode a))` polynomial, and executes a *monomial cancelation*. If exists in the input polynomial a *positive* and *negative* occurrence of any monomial, both occurrences of the monomial are removed from the polynomial.

4. Finally, we add an extra constraint `delta > 0` with `and (gr delta zero) p3`. We do this to force the solver, to give `delta` a value greater than zero.

Now the constraint is ready to be sent to the constraint solver.

5.4 SMT Solver

Once a set of simple constraints has been generated and collected, we can proceed with the last step of the process, and solve the constraints. In our tool, we use the *barcelogicsNIA* solver

http://www.cs.upc.edu/~albert/nonlinear.html which is an SMT-solver. In order to call the solver (using the `Solver.SolverSMTExt` module from MU-TERM) we use function `solveSMT2`, that takes the constraint to be solved, transforms it into an appropriate (SMT-LIB) format, and sends it to Barcelogics to solve the inequations.

If the returned result is *SAT* (for *Satisfiable*) the solver also returns the *value* for each *variable* in the constraint. Otherwise, Barcelogics only returns the word *UNSAT*.

The returned results give information about the given value for the *parametric coefficients* which are used to build the constraints during the transformation process described in the previus sections. Each variable (or *parametric coefficient*) is given a solution in the SMT solver output which is described as follows:

$$[\mathtt{V_{name}} : \mathtt{Sort},\ (\mathtt{value}, \mathtt{type})]$$

where `value` is the numeric value associated by the solver to variable $\mathtt{V_{name}}$. Since the solver returns *Integer* values for all answers, the `type` parameter is 1 in all cases.

6 Conclusions and Future Work

6.1 Conclusions

Automatic Program Analysis and Verification is a field in expansion. As new theories are defined, there will always be a need for checking properties about these theories (or expressed by means of these theories) in an automatic way. We think that tools like AGES, created for this project, can be helfpful for researchers and practitioners to improve their ability to deal with software analysis and development.

Our tool is able to read a program, as a Maude specification, to parse and transform it into an Order-Sorted First-Order theory. Also, the goals defined for the targeted analysis problem, are parsed from a text with similar, Maude-like format into another Order-Sorted First-Order theory. The tool then generates a model for them based on interpretating sorts as convex domains and function symbols as linear polynomial interpretations. In this way, the original theory is transformed into a set of constraints, which are solved with an external tool, and the results of that process, are used to assemble a model for the initial program and goals.

An interesting aspect of this work is the possibility of sharing with the MU-TERM tool in *both* directions. To achieve future compatibility between AGES (and their implemented processes) and MU-TERM, both tools use common data types. In this way, we can furnish MU-TERM with new features that will increase the power of the tool.

About the developer skills acquired, some people will discuss about the *benefits* of the functional programming. For this project, using Haskell as a programming language has been an advantage, because of code reuse. The learning curve of Haskell is very steep, but once we change our *imperative* way to analyze programming problems, it becomes easier to figure out *solutions* encoded as *functions*.

6.2 Future Work

AGES has been created using MU-TERM as base framework, but it is an independent application. The logical next step will be the integration of modules created for AGES into MU-TERM, to enable the verification of the *termination* property, using the *convex* *domains* that are now available in AGES (see [11, 13] for further motivation).

The current release of AGES has a lot of improvements to be made. The use of more general convex domains (of bigger dimension) and the convex matrix interpretations proposed in [11] (that extend linear polynomial interpretations by using *matrices* as coefficients of the linear expressions, see Section 2.2.2) could be a first step in improving its performance. Changes like these, will improve the generation of logical models and the validation of handcrafted solutions.

Also, as it was mentioned before, this tool has no time constraint like other analysis and verification tools. When used for research, tools like AGES, can be used in long-run experiments (that was the main reason to create a web-based and a command line version of the tool), enhancing the arsenal of available software for automatic verification and analysis. But improving efficiency and achieving a fast analysis tool is also an important subject for future work.

Bibliography

[1] B. Alarcón, R. Gutiérrez, S. Lucas, R. Navarro-Marset. Proving Termination Properties with MU-TERM. In M. Johnson and D. Pavlovic, editors, *Proc. of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST'10*, LNCS 6486:201-208, Springer-Verlag, 2011.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All About Maude – A High-Performance Logical Framework. Lecture Notes in Computer Science 4350, 2007.

[3] E. Contejean, C. Marché, A.-P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325-363, 2006.

[4] F. Durán, S. Lucas, C. Marché, J. Meseguer, X. Urbain, Proving Operational Termination of Membership Equational Programs, Higher-Order and Symbolic Computation 21(1-2):59–88, 2008.

[5] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp,, R. Thiemann and H.Zankl. Maximal Termination in Proc. of RTA'08, LCNS 5117:110-125, Springer-Verlag, Berlin, 2008

[6] J. Goguen and J. Meseguer. Models and Equality for Logical Programming. In *Proc. of TAPSOFT'87*, LNCS 250:1-22, Springer-Verlag, 1987.

[7] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[8] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current trends in Programming Methodology*, pages 80-149, Prentice Hall, 1978.

[9] W. Hodges. A shorter model theory. Cambridge University Press, 1997.

[10] S. Lucas. Automatic generation of logical models for order-sorted first-order theories InProgramAnalysis. In M. Navarro, editor, *Proc. of the XV Jornadas sobre Programación y Lenguajes, PROLE'15*, to appear, 2015.

[11] S. Lucas and J. Meseguer. Models for Logics and Conditional Constraints in Automated Proofs of Termination. In G.A. Aranda-Corral and F.J.

Martín-Mateos, editors, Proc. of *the 12th International Conference on Artificial Intelligence and Symbolic Computation, AISC'14*, LNAI 8884:7-18, 2014.

[12] S. Lucas. Polynomials over the Reals in Proofs of Termination: from Theory to Practice. *RAIRO Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[13] S. Lucas and J. Meseguer. Proving Operational Termination Of Declarative Programs In General Logics. In O. Danvy, editor, *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming , PPDP'14*, pages 111-122, ACM Press, 2014. DOI: 10.1145/2643135.2643152

[14] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems - Safety. Springer-Verlag, 1995.

[15] J. Meseguer. General Logics. In *Logic Colloquium'87*, pages 275-329, North-Holland, 1989.

[16] F. Nielson, H.R. Nielson, and C. Hankin. Principles of Program Analysis. Springer-Verlag, Berlin, 1999.

[17] M.O. Rabin. Decidable Theories. In J. Barwise, editor, Handbook of Mathematical Logic, North-Holland, 1977.

[18] A. Schrijver. Theory of linear and integer programming. John Wiley & sons, 1986.