

UNIVERSITÀ DEGLI STUDI DI PISA

Dipartimento di Informatica

Laurea Specialistica In Informatica

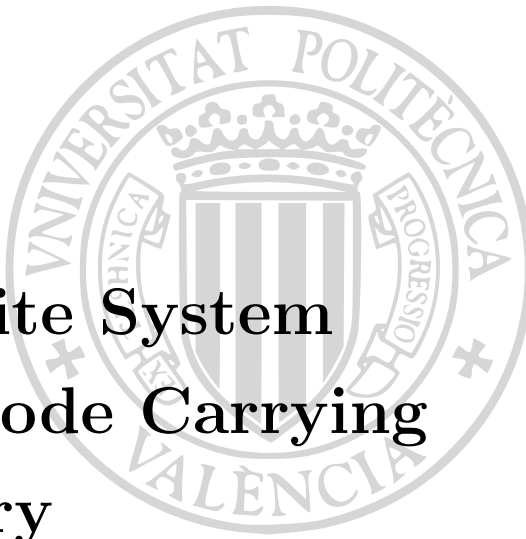
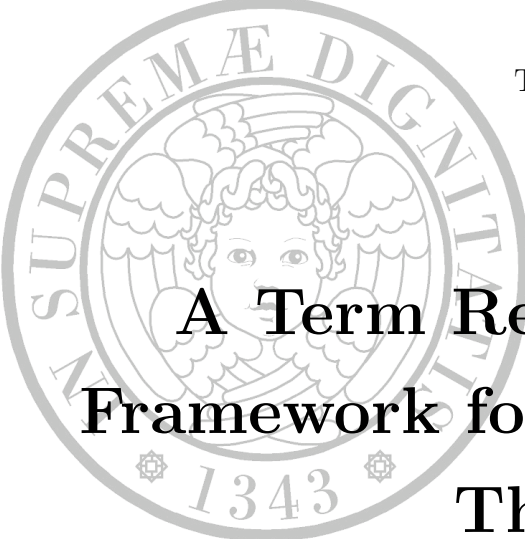
&

UNIVERSIDAD POLITÉCNICA DE VALENCIA

Depto. Sistemas Informáticos y Computación

Ingeniería Informática

THESIS



**A Term Rewrite System
Framework for Code Carrying
Theory**

CANDIDATE:

Paolo Picci

SUPERVISORS:

María Alpuente Frasnado

Giorgio Levi

July, 2011

*Dipartimento di Informatica
Università di Pisa
Via F. Buonarroti, 2
I-56127 Pisa, Italia*

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

Contents

1	Introduction	1
1.1	Plan of the Thesis.	2
2	Background on Term Rewriting	5
2.1	Terminology and definitions	5
2.2	Term Rewriting	6
3	The Rewrite Framework	9
3.1	Introduction	9
3.2	Narrowing in Rewriting Logic	10
3.3	Transforming Rewrite Theories	12
3.3.1	Definition Introduction	12
3.3.2	Definition Elimination	14
3.3.3	Folding	14
3.3.4	Unfolding	16
3.3.5	Abstraction	17
3.4	Program Semantics and Correctness of the transformation system	18
4	Rewrite on Code Carrying Theory	21
4.1	Background on CCT	21
4.1.1	Code-Carrying Theory Steps	23
4.2	Program Synthesis and CCT	25
4.2.1	Certificate	25
4.2.2	Steps on new framework	26

4.2.3	Case Study: a tail recursive function	28
5	Security attacks and extension of the framework	31
5.1	Security	31
5.1.1	Hacking the certificate	34
5.2	Checking the Certificate	37
5.2.1	Preconditions	38
6	Pyconnect	43
6.1	Description and Motivation	43
6.2	Configuration	44
6.2.1	Configuration example	45
6.3	Functions	47
6.3.1	Channels and sending groups	48
6.3.2	Pyconnect commands	49
6.4	Extending the software	50
7	Putting all together	53
7.1	Implementation	53
7.1.1	Features	55
8	Conclusions	59
	Bibliography	63
	List of Figures	63

Acknowledgments

My most heartfelt thanks to my supervisors: Doctor Maria Alpuente who has monitored constantly and patiently the writing of this thesis, Doctor Giorgio Levi who has encouraged me and given his support to its development abroad. Gisella for her help with English. I would also like to thank my parents for their financial support and - above all - their invaluable advice, and my sister who has proved to be a great friend. Finally, I thank all the friends who have helped me, directly or indirectly, to grow both professionally and as a person. Thanks to all.

Chapter 1

Introduction

The problem of data security is a fundamental aspect in any sector, and the growing ubiquity of mobile and distributed systems has accentuated the problem. Mobile code is software that is transferred between systems and executed on a local system without explicit installation by the recipient, even if it is delivered through an insecure network or retrieved from an untrusted source. During delivery, the code may be corrupted or a malicious cracker could change the code damaging the entire system. Potential problems can be summarized as problems related to security, allowing access to data or system resources which were not previously authorized, illegal or unlawful activities on the data, or functional incorrectness, that arises when the provided code fails to satisfy a necessary connection between its input and output. Code-Carrying Theory (CCT) [18, 19] is one of the technologies aimed at solving these problems. The idea of CCT is based on proof-based program synthesis, where a set of axioms that define functions are provided by the code producer together with suitable proofs guaranteeing that defined functions obey certain requirements. The form of the function-defining axioms is such that it is easy to extract executable code from them. Thus, all that has to be transmitted from the producer to the consumer is a theory (a set of axioms and theorems) and a set of proofs of the theorems. There is no need to transmit code explicitly. A basic implementation of the CCT methodology that uses a Fold/Unfold transformation framework for rewrite theories, and that reduces

the burden on the code producer is done in Rewriting logic [8]. Rewriting logic is efficiently implemented in the high-performance functional language Maude [10]. The purpose of this thesis is to extend and improve the framework for the CCT turning it into a stable and usable tool for Maude. The implementation is written in Maude itself, Python and some scripts in Bash. This thesis describe the general architecture of the system and the technical aspects that make it all modular and extensible.

1.1 Plan of the Thesis.

The thesis is organized as follows:

- In Chapter 2, we provide the necessary notation and preliminary definitions about the term rewriting formalism that will be used in this thesis.
- In Chapter 3, we recall a Fold/Unfold based transformation framework for rewriting logic theories that we apply to implement the Code Carrying Theory (CCT) system.
- In Chapter 4, we describe the overall structure of the Code Carrying Theory (CCT) system, and discusses how the framework described in Chapter 3 can be embedded into the system.
- In Chapter 5, we analyze the security of the framework and analyze a few examples of attacks that witness its weaknesses. Then we discuss how it is possible to prevent the attacks with the introduction of a suitable procedure for checking the certificates.
- In Chapter 6, we describe the architecture of Pyconnect, a software system for connecting and communicating different processes through

only one system shell.

- In Chapter 7, we describe the resulting refined framework for CCT extended for Certificate Checking, how it is assembled and how it works.
- In Chapter 8, we present our conclusions and discuss some lines for future work.

Chapter 2

Background on Term Rewriting

In this chapter, we provide the basic notation and terminology about rewriting logic and term rewriting system, that are used in this thesis.

2.1 Terminology and definitions

We consider an *order-sorted signature* Σ , with a finite poset of sorts (S, \leq) . We assume an S -sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. A variable $x \in \mathcal{V}$ of sort s is denoted by $x :: s$, while by $f :: s_1 \dots s_n \mapsto s$ we represent the signature of the operator $f \in \Sigma$ of arity n and type s . $\mathcal{T}_\Sigma(\mathcal{V})_s$ and \mathcal{T}_{Σ_s} are respectively the sets of terms and ground terms of sort s . We write $\mathcal{T}_\Sigma(\mathcal{V})$ and \mathcal{T}_Σ for the corresponding term algebras. The set of variables occurring in a term t is denoted by $\text{Var}(t)$. For simplicity we write $\overline{o_n}$ for the *list* of syntactic objects o_1, \dots, o_n .

Positions are represented by sequences of natural numbers. The empty sequence Λ denotes the root position of a term. Given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t that are rooted by symbols in S . Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. $t|_p$ denotes the *subterm* of t at position p , and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s . Syntactic equality is represented by \equiv .

A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \dots\}$ is a mapping from the set of vari-

ables \mathcal{V} into the set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ satisfying the following conditions: (i) $x_i \neq x_j$, whenever $i \neq j$, (ii) $x_i\sigma = t_i$, $i = 1, \dots, n$ and (iii) $x\sigma = x$, for all $x \in \mathcal{V} \setminus \{x_1, \dots, x_n\}$. By ε we denote the *empty* substitution. A substitution θ is *more general* than a substitution σ , in symbols $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ . Given two terms s and t , a *unifier* for s and t is a substitution σ such that $s\sigma = t\sigma$. By $mgu(s, t)$ we denote the *most general unifier* for s and t . An *instance* of a term t is defined as $t\sigma$, where σ is a substitution. The identity substitution is denoted by id .

2.2 Term Rewriting

Term rewriting systems (TRS) provide an adequate computational model for functional languages[15][7]. In this section, we provide a brief overview of such a model following the standard framework of term rewriting.

An (*order-sorted*) *equational theory* is a pair $E \equiv (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a set of equations of the form $l = r$, where $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$, $l \notin \mathcal{V}$, and B is a set of algebraic axioms such as the associativity, commutativity, and identity declared for the different defined functions. We assume Σ can be always considered as the disjoint union $\Sigma \equiv \mathcal{C} \uplus \mathcal{D}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{D}$, called *defined functions*, each one having a fixed arity, where $\mathcal{D} \equiv \{f \mid f(\bar{t}) = r \in \Delta\}$ and $\mathcal{C} \equiv \Sigma - \mathcal{D}$. Then $\mathcal{T}_\mathcal{C}(\mathcal{V})$ is the set of constructor terms. Given an equation $l = r$, terms l and r are called the *left-hand side* (or *lhs*) and the *right-hand side* (or *rhs*) of the equation, respectively, and $\text{Var}(r) \subseteq \text{Var}(l)$.

The equations in an equational theory E are considered as simplification rules by using them only in the left to right direction, so for any term t , by repeatedly applying the equations as simplification rules, we eventually reach a term to which no further equations apply. The result is called the *canonical form* of t w.r.t. E . This is guaranteed by the fact that E is required to be terminating and Church-Rosser [4]. The set of equations in Δ together with the equational axioms of B in an equational theory E induce a congruence relation on the set of terms $\mathcal{T}_\Sigma(\mathcal{V})$ which is usually denoted by $=_E$. E is

2.2. Term Rewriting

a presentation or axiomatization of $=_E$. In abuse of notation, we speak of the equational theory E to denote the theory axiomatized by E . Given a canonical equational theory E , we say that a substitution σ is a E -unifier of two generic terms t and t' if $t\sigma$ and $t'\sigma$ are both reduced to the same canonical form modulo the equational theory (in symbols $t\sigma =_E t'\sigma$).

A (*order-sorted*) *rewrite theory* is a triple $\mathcal{R} \equiv (\Sigma, \Delta \cup B, R)$, where R is a set of rewrite rules of the form $l \rightarrow r$, where $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$, $l \notin \mathcal{V}$, and Σ is the pairwise disjoint union $\mathcal{D}_1 \uplus \mathcal{D}_2 \uplus \mathcal{C}$ such that $(\mathcal{D}_1 \uplus \mathcal{C}, \Delta \cup B)$ is an order-sorted equational theory and $\mathcal{D}_2 \equiv \{f \mid f(\bar{t}) \rightarrow r \in R\}$. Symbols in \mathcal{D}_2 are called *defined symbols* as well as those in \mathcal{D}_1 , with the only difference that the former are defined in rewrite rules, while the latter in the equational theory. We omit Σ when no confusion can arise. Given a rule $l \rightarrow r$, terms l and r are called the *left-hand side* (or *lhs*) and the *right-hand side* (or *rhs*) of the rule, respectively, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. An equation of the form $t = t'$ or a rule of the form $t \rightarrow t'$ are said to be:

- (1) *Non-erasing*, if $\mathcal{V}ar(t) = \mathcal{V}ar(t')$.
- (2) *Sort preserving*, if for each substitution σ , we have $t\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$ if and only if $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$.
- (3) *Sort decreasing*, if for each substitution σ , $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$ implies $t\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_s$.
- (4) *Left (or right) linear*, if t (*resp.* t') is *linear*, i.e., no variable occurs in the term more than once. It is called *linear* if both t and t' are linear.

A set of equations/rules is said to be non-erasing, or sort decreasing, or sort preserving, or (left or right) linear, if each equation/rule in it is so.

An equational theory (*resp.* rewrite theory) is said to be *conditional* if its equations (*resp.* rules) are of the form $(l = r \text{ if } c)$ (*resp.* $l \rightarrow r \text{ if } c$), where c is a term representing the condition. Moreover, *labels* may be associated with equations and rules in order to easily identify them, in the form $(\text{label} : l = r)$ or $(\text{label} : l \rightarrow r)$.

We define the *one-step rewrite relation* on $\mathcal{T}_\Sigma(\mathcal{V})$ as follows: $t \rightarrow_R t'$ if there is a position $p \in O_\Sigma(t)$, a rule $l \rightarrow r$ in R , and a substitution σ such

that $t|_p \equiv l\sigma$ and $t' \equiv t[r\sigma]_p$. An instance $l\sigma$ of a rule $l \rightarrow r$ is called a *redex*. A term t without redexes is called *normal form*. Let $R \equiv l \rightarrow r$ be a rule in a given rewrite theory and $t \rightarrow_R t'$ be a rewrite step reducing a redex at position $p \in O_\Sigma(t)$.

The relation $\rightarrow_{R/E}$ for rewriting modulo E is defined as $=_E \circ \rightarrow_R \circ =_E$. Let $\rightarrow \subseteq A \times A$ be a binary relation on a set A . We denote the transitive closure by \rightarrow^+ , the reflexive and transitive closure by \rightarrow^* , and the rewrite up to normal form by $\rightarrow^!$.

A rewrite theory is *sufficiently complete* if enough rules/equations have been specified, so that functions of the theory are fully defined on all relevant data.

Example 1 Consider the following rewrite theory $(\Sigma, \Delta \cup B, R)$ such that $\mathcal{C} \equiv \{b, c, e\}$, $\mathcal{D}_1 \equiv \{a, d\}$, $\mathcal{D}_2 \equiv \{f\}$, $\Delta \equiv \{a = b, d = e\}$, $R \equiv \{f(b, c) \rightarrow d\}$ where B contains the commutativity axiom for f . Then we can R/E -rewrite term $f(c, a)$ to e by means of the following R/E rewrite sequence $f(c, a) =_\Delta f(c, b) =_B f(b, c) \rightarrow_R d =_\Delta e$. ■

We say that a rewrite theory $\mathcal{R} \equiv (\Sigma, \Delta \cup B, R)$ is *terminating* w.r.t. $\rightarrow_{R/E}$, if there exists no infinite rewrite sequence $t_1 \rightarrow_{R/E} t_2 \rightarrow_{R/E} \dots$. A rewrite theory is *confluent* w.r.t. $\rightarrow_{R/E}$ if, for all terms s, t_1, t_2 , such that $s \rightarrow_{R/E}^* t_1$ and $s \rightarrow_{R/E}^* t_2$, there exists a term t s.t. $t_1 \rightarrow_{R/E}^* t$ and $t_2 \rightarrow_{R/E}^* t$.

Chapter 3

The Rewrite Framework

In this chapter, we show the theoretical framework for transformation rules for rewrite theories used to implement Code Carrying Theory [18, 19] and presented in [8, 5, 6], explaining how it's composed and how it works.

In the first part, we formalize the operation of *narrowing* that is used by some functions of the framework. After that, we proceed to illustrate the basic steps and operations that are used and compose the transformations system, analyzing the preconditions and postconditions that ensure the completeness of the framework. Finally, we provide some examples of transformation sequences, that will make more clear to the reader the structure and the powerfulness of the system.

3.1 Introduction

Many transformation systems for program optimization, program synthesis, and program specialization are based on fold/unfold transformations, also know as the rules+ strategies approach. In this chapter, we discuss a fold/unfold-based transformation framework for rewriting logic theories which is based on narrowing, an generalization of term rewriting which replaces pattern matching by unification. To the best of our knowledge, this is the first fold/unfold transformation framework which allows one to deal with

functions, rules, equations, sorts, and algebraic laws (such as commutativity and associativity).

When performing program transformation, we end up with a final program which is semantically equal to the initial one. During the transformation process, we need strategies (such as composition and tupling) which guide the application of the transformation rules and allow us to derive programs with improved performance, which can be done in a semi-automatic way. The process for obtaining a correct and efficient program can be split in two phases, which may be performed by different actors: the first phase is to write an initial maybe inefficient program whose correctness can be easily shown by hand or by automatic tools; in the second phase, the actor transforms the initial program by applying the rules of the framework to derive a more efficient one.

We consider possibly non-confluent and non-terminating rewriting logic theories, and the operations for transforming these rewriting logic theories, with a strategy to restore completeness defined in [8], that preserves the rewriting logic semantics of the original theory.

3.2 Narrowing in Rewriting Logic

Considering the rewrite relation $\rightarrow_{R/E}$ introduced in Chapter 2, since E -congruence classes can be infinite, $\rightarrow_{R/E}$ -reducibility is undecidable in general. One way to overcome this problem is to implement R/E -rewriting by a combination of rewriting using oriented equations and rules [20].

We define the relation $\rightarrow_{\Delta,B}$ on $\mathcal{T}_\Sigma(\mathcal{V})$ as follows: $t \rightarrow_{\Delta,B} t'$ if there is a position $p \in O_\Sigma(t)$, $l = r$ in Δ , and a substitution σ such that $t|_p =_B l\sigma$ and $t' = t[r\sigma]_p$. The relation $\rightarrow_{R,B}$ is similarly defined, and we define $\rightarrow_{R \cup \Delta, B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. The idea is to implement $\rightarrow_{R/E}$ using $\rightarrow_{R \cup \Delta, B}$. For this approach to be correct and complete, we need the following assumptions [16]. We assume the following properties on $E = \Delta \cup B$.

- (i) B is *non-erasing*, and *sort preserving*.
- (ii) B has a finitary and complete unification algorithm, which implies that

3.2. Narrowing in Rewriting Logic

B -matching is decidable, and $\Delta \cup B$ has a complete (but not necessarily finite) unification algorithm.

- (iii) Δ is *sort decreasing*, and *confluent and terminating modulo B* .
- (iv) $\rightarrow_{\Delta, B}$ is *coherent with B* , i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{\Delta, B}^+ t_2$ and that $t_1 =_B t_3$ implies $\exists t_4, t_5$ such that $t_2 \rightarrow_{\Delta, B}^* t_4$, $t_3 \rightarrow_{\Delta, B}^+ t_5$, and $t_4 =_E t_5$.
- (v) $\rightarrow_{R, B}$ is *E -consistent with B* , i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{R, B} t_2$ and that $t_1 =_B t_3$ implies $\exists t_4$ such that $t_3 \rightarrow_{R, B} t_4$, and $t_2 =_E t_4$.
- (vi) $\rightarrow_{R, B}$ is *E -consistent with $\rightarrow_{\Delta, B}$* , i.e., $\forall t_1, t_2, t_3$, we have that $t_1 \rightarrow_{R, B} t_2$ and that $t_1 \rightarrow_{\Delta, B}^* t_3$ implies $\exists t_4, t_5$ such that $t_3 \rightarrow_{\Delta, B}^* t_4$, $t_4 \rightarrow_{R, B} t_5$, and $t_5 =_E t_2$.

Narrowing [9],[12] generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification (at non-variable positions) instead of matching in order to (nondeterministically) reduce a term. The narrowing mechanism has a number of important applications including automated proofs of termination, execution of functional-logic programming languages, partial evaluation, verification of cryptographic protocols and equational unification. The narrowing relation for rewriting logic theories is defined as follows [16].

Definition 1 ($R \cup \Delta, B$ -Narrowing) Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be an order-sorted rewrite theory satisfying properties (i) - (vi) above. The $R \cup \Delta, B$ -narrowing relation on $\mathcal{T}_\Sigma(\mathcal{V})$ is defined as $t \rightsquigarrow_{\sigma, p, R \cup \Delta, B} t'$ if there exist $p \in O_\Sigma(t)$, a rule $l \rightarrow r$ or equation $l = r$ in $R \cup \Delta$, and σ , which is a B -unifier of $t|_p$ and l such that $t' = (t[r]_p)\sigma$. $t \rightsquigarrow_{\sigma, p, R \cup \Delta, B} t'$ is also called a $R \cup \Delta, B$ -narrowing step.

Example 2 Consider the following rewrite theory $(\Sigma, \Delta \cup B, R)$ such that $\mathcal{C} = \{b, c, e\}$, $\mathcal{D}_1 = \{a, d\}$, $\mathcal{D}_2 = \{f\}$, $\Delta = \{a = b, d = e\}$, $R = \{f(x, f(y, b)) \rightarrow d\}$ where B contains the commutativity axiom for f . Then we can perform the narrowing step $f(f(w, z), c) \rightsquigarrow_{\sigma, \Lambda, R \cup \Delta, B} d$, with $\sigma = \{x/b, z/b\}$, since for

the commutativity of f we have that $f(f(w, z), c)\{z/b\} =_B f(x, f(y, b))\{x/c\}$.

■

When it is clear from the context, we omit $R \cup \Delta, B$ from the narrowing relation. Narrowing derivations are denoted by $t_0 \rightsquigarrow_\sigma^* t_n$, which is a shorthand for the sequence of narrowing steps $t_0 \rightsquigarrow_{\sigma_1, p_1} \dots \rightsquigarrow_{\sigma_n, p_n} t_n$ with $\sigma = \sigma_n \circ \dots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$).

3.3 Transforming Rewrite Theories

In this section, we recall the fold/unfold transformation framework of PEPM 2010 by introducing the transformation rules over rewrite theories.

A *transformation sequence* of length k for a rewrite theory $(\Sigma, \Delta \cup B, R)$ is a sequence $\mathcal{R}_0, \dots, \mathcal{R}_i, \mathcal{R}_{i+1}, \dots, \mathcal{R}_k$, $k \geq 0$, where each \mathcal{R}_j , $0 \leq j \leq k$ is a rewrite theory, such that

- $\mathcal{R}_0 = (\Sigma, E_0, R_0)$, with $E_0 = (\Delta \cup B)$ and $R_0 = R$.
- For each $0 \leq j < i$, $\mathcal{R}_{j+1} = (\Sigma, \Delta_{j+1} \cup B, R_0)$ is derived from \mathcal{R}_j by an application of a transformation rule on the equation set Δ_j .
- For each $i \leq j < k$, $\mathcal{R}_{j+1} = (\Sigma, E_i, R_{j+1})$ is derived from \mathcal{R}_j by an application of a transformation rule on the rule set R_j .

The *transformation rules* are Definition Introduction, Definition Elimination, Folding, Unfolding, and Abstraction, which are defined as follows.

3.3.1 Definition Introduction

We can obtain program \mathcal{R}_{k+1} by adding to \mathcal{R}_k a set of new equations (*resp.* rules), defining a new symbol f called *eureka*. We consider equations (*resp.* rules) of the form $f(\bar{t}_i) = r_i$ (*resp.* $f(\bar{t}_i) \rightarrow r_i$), such that:

- (1) f is a function symbol which does not occur in the sequence $\mathcal{R}_0, \dots, \mathcal{R}_k$ and is declared by $f : s_1 \dots s_n \rightarrow s$ $[\mathbf{Ax}]$, where s_1, \dots, s_n, s are sorts declared in \mathcal{R}_0 and \mathbf{Ax} are equational attributes.

3.3. Transforming Rewrite Theories

- (2) $t_i \in \mathcal{T}_C(\mathcal{V})$, and $\mathcal{V}ar(\bar{t}_i) = \mathcal{V}ar(r_i)$, for all i — i.e., the equations/rules are *non-erasing*.
- (3) Every defined function symbol occurring in r_i belongs to \mathcal{R}_0 .
- (4) The set of new equations (*resp.* rules) are left linear, sufficiently complete and non overlapping. For rules we require also right linearity.

In general, the main idea consists of introducing new auxiliary function symbols which are defined by means of a set of equations/rules whose bodies contain a subset of the functions that appear in the right-hand side of an equation/rule that appears in \mathcal{R}_0 , whose definition is intended to be improved by subsequent transformation steps. The non overlapping property and the left-linearity ensure confluence of eureka, which is needed to preserve the completeness of the fold operation and will be discussed later. Sufficient completeness is needed to ensure the completeness of unfolding and will be discussed later. Right-linearity on rules is needed to ensure narrowing completeness [16], and left-linearity is also needed to preserve the right-linearity of rules when doing folding. Consider, for instance, the folding of rule $f(x) \rightarrow g(x)$ using the (non left linear) eureka $new(x, x) \rightarrow g(x)$, which would produce a new rule $f(x) \rightarrow new(x, x)$, which is not right-linear.

Note that, once a transformation is applied to a eureka, the obtained equation/rule is not considered to be a eureka anymore. As we will see later, this is important for the folding operation, since we can only fold non-eureka equations/rules using eureka ones.

The *non-erasing* condition is a standard requirement that avoids the creation of equations/rules with extra-variables when performing folding steps. Consider, for instance, the folding of equation $f(x) = g(x)$ using the (erasing) eureka $new(x, y) = g(x)$, which would produce a new equation $f(x) = new(x, y)$ containing an extra variable in its right-hand side (thus an illegal equation).

3.3.2 Definition Elimination

Let \mathcal{R}_k be the rewrite theory $(\Sigma_k, \Delta_k \cup B_k, R_k)$. We can obtain program \mathcal{R}_{k+1} by deleting from program \mathcal{R}_k ,

- all equations that define the functions f_0, \dots, f_n , say Δ^f , such that f_0, \dots, f_n do not occur either in \mathcal{R}_0 or in $(\Sigma_k, (\Delta_k \setminus \Delta^f) \cup B_k, R_k)$.
- all rules that define the functions f_0, \dots, f_n , say R^f , such that f_0, \dots, f_n do not occur either in \mathcal{R}_0 or in $(\Sigma_k, \Delta_k \cup B_k, R_k \setminus R^f)$.

Note that the deletion of the equations/rules that define a function f implies that no function calls to f are allowed afterwards. However, subsequent transformation steps (in particular, folding steps) might introduce those deleted functions in the rhs's of the equations/rules, thus producing inconsistencies in the resulting programs. To avoid this, we forbid any folding step after a definition elimination has been performed (this generally boils down to postpone all elimination steps to the end of the transformation sequence).

3.3.3 Folding

Roughly speaking the Folding operation is the replacement of some piece of code by an equivalent function call. Let $F \in \mathcal{R}_k$ be an equation (the "folded equation") of the form $(l = r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be an equation (the "folding equation") of the form $(l' = r')$, such that $r|_p =_{B_k} r'\sigma$ for some position $p \in O_\Sigma(r)$ and substitution σ . Note that, since we transform the equations of an equational theory, we consider here the congruence relation $=_{B_k}$ modulo the equational axioms B_k (assuming an empty equation set). This is because we cannot consider a congruence modulo an equational theory which is being modified. Moreover, the following conditions must be satisfied:

- (1) F is not a eureka.
- (2) F' is a eureka.

3.3. Transforming Rewrite Theories

- (3) The substitution σ is sort decreasing, i.e, if $x \in \mathcal{V}_s$, then $x\sigma \in \mathcal{T}_\Sigma(\mathcal{V})_{s'}$ such that $s' \leq s$.
- (4) Let $l' = f(\overline{t_n})$ and $r|_p = e$ and let $f(\overline{t_n})$ and e have type s_f and s_e , respectively; then $s_f \leq s_e$.

Then, we can obtain program \mathcal{R}_{k+1} from program \mathcal{R}_k by replacing F with the new equation $(l = r[l'\sigma]_p)$.

Folding can be applied to rules whenever the transformation of the equational theory has been completed. To fold rules we proceed as follows. Let $F \in \mathcal{R}_k$ be a rule (the "folded rule") of the form $(l \rightarrow r)$, and let $F' \in \mathcal{R}_j$, $0 \leq j \leq k$, be a rule (the "folding rule") of the form $(l' \rightarrow r')$, such that $r|_p =_{E_k} r'\sigma$ for some position $p \in O_\Sigma(r)$ and substitution σ , fulfilling conditions (1) - (4) above. Then, we can obtain program \mathcal{R}_{k+1} from program \mathcal{R}_k by replacing F with the new rule $(l \rightarrow r[l'\sigma]_p)$.

The need for conditions (1) and (2) is twofold. These conditions forbid self-folding, that is, a folding operation with $F = F'$, thus a rule with the same left and right-hand side cannot be produced, which may introduce infinite loops on derivations and destroy the correctness properties of the transformation system. These conditions also forbid the folding of a eureka, which is meaningless as illustrated in the following example.

Example 3 Consider the following two rules:

$$\begin{array}{ll} new \rightarrow f & \text{(eureka)} \\ g \rightarrow f & \text{(non-eureka)} \end{array}$$

Without conditions (1) and (2), a folding of the eureka rule would be possible, obtaining the new rule $(new \rightarrow g)$, which is nothing more than a redefinition of the symbol new . Since transformation rules aim at optimizing the original program with the support of eureka, a folding over a eureka is meaningless or even dangerous. ■

Finally, conditions (3) and (4) ensure the sort compatibility of both the applied substitutions and the term that is inserted into the folded equation/rule right-hand side. We can see an example of Folding operation.

Example 4 Consider the following rewrite theory:

$\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where $\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R and

$$\begin{array}{ll}
 R : & R' : \\
 f(a, b) \rightarrow g(a, b) & f(a, b) \rightarrow g(h(a), b) \\
 f(x, y) \rightarrow g(x, y) & f(x, y) \rightarrow g(x, y) \\
 g(a, x) \rightarrow a & g(a, x) \rightarrow a \\
 g(b, x) \rightarrow b & g(b, x) \rightarrow b \\
 h(a) \rightarrow a & h(a) \rightarrow a
 \end{array}$$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from \mathcal{R} by applying a fold step to the rule $f(a, b) \rightarrow g(a, b)$ using the eureka $h(a) \rightarrow a$. ■

3.3.4 Unfolding

Unfolding is essentially the replacement of a call by its body, with appropriate substitutions. Let us introduce formally the Unfolding operation:

Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a program and let F be an equation (*resp.* rule) of the form $l = r$ (*resp.* $l \rightarrow r$) in \mathcal{R} . We obtain a new program from \mathcal{R} by replacing F with the set of equations (*resp.* rules)

$$\{l\sigma = r' \mid r \rightsquigarrow_{\sigma, \Delta, B} r' \text{ is a } \Delta, B \text{ narrowing step}\}$$

$$\{l\sigma \rightarrow r' \mid r \rightsquigarrow_{\sigma, R \cup \Delta, B} r' \text{ is a } R \cup \Delta, B \text{ narrowing step}\}$$

Since we consider rewrite theories where defined symbols are allowed to be arbitrarily nested in left-hand sides of rules, rule unfolding may cause the loss of completeness for the transformed program w.r.t. the semantics of the original one. Let us illustrate this problem by means of an example.

Example 5 Consider the following rewrite theory $\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where

$\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R and

$$\begin{array}{ll}
 R : & R' : \\
 1. & g_1(x) \rightarrow x \qquad g_1(x) \rightarrow x \\
 2. & h(x) \rightarrow 0 \qquad h(x) \rightarrow 0 \\
 3. & h(g_1(x)) \rightarrow 1 \qquad h(g_1(x)) \rightarrow 1 \\
 4. & f(x) \rightarrow g_1(x) \\
 5. & \mathbf{f}(\mathbf{x}) \rightarrow \mathbf{x}
 \end{array}$$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from \mathcal{R} by applying an unfolding step over rule 4 in R , through the narrowing step $g_1(x) \rightsquigarrow_{\varepsilon} x$. Term $h(f(0))$ can be rewritten in \mathcal{R} to the normal forms 0 or 1 by means of the rewrite sequences $h(f(0)) \rightarrow_4 h(g_1(0)) \rightarrow_1 h(0) \rightarrow_2 0$, and $h(f(0)) \rightarrow_4 h(g_1(0)) \rightarrow_3 1$, respectively. The only possible rewrite sequences from $h(f(0))$ in \mathcal{R}' are $h(f(0)) \rightarrow_2 0$, and $h(f(0)) \rightarrow_5 h(0) \rightarrow_2 0$, thus we miss normal form 1. In fact, symbol g_1 is needed for rule 3 to be applied, and function f provides that occurrence of g_1 needed to reach the normal form 1. However, the unfolding of rule 4 forces the occurrence of symbol g_1 to be evaluated, and, hence, that rewrite sequence is no longer available in \mathcal{R}' . ■

In [8] a procedure to restore completeness of the new program \mathcal{R}' is proposed, together with a proof of soundness and a methodology for optimizing that procedure, which are outside the scope of this thesis.

3.3.5 Abstraction

The set of rules presented so far constitute the core of our transformation system; however let us mention another useful rule, called Abstraction, which can be simulated in our settings by applying appropriate definition introduction and folding steps. This rule is usually required to implement tupling, and it consists of replacing, by a new function, multiple occurrences of the same expression e in the right-hand side of an equation/rule. For instance, consider the following equation

$$double_sum(x, y) = sum(sum(x, y), sum(x, y))$$

where $e = \text{sum}(x, y)$. The equation can be transformed into the following pair of equations

$$\begin{aligned} \text{double_sum}(x, y) &= \text{ds_aux}(\text{sum}(x, y)) \\ \text{ds_aux}(z) &= \text{sum}(z, z) \end{aligned}$$

These equations are generated from the original one by a definition introduction of the eureka ds_aux and then by folding the original equation by means of the newly generated eureka.

Note that the abstraction rule applies to equations or rules which are not right-linear, since the same expression e occurs more than once in their rhs. Since we ask for rules to be right-linear for the completeness of the narrowing relation, we may think to use the abstraction rule to preprocess rewrite rules in order to try to make them right-linear.

3.4 Program Semantics and Correctness of the transformation system

We now provide a definition of the considered program semantics.

Definition 2 (Program Semantics) *Given a program $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, the semantics of ground reducts of \mathcal{R} is the set $\text{gred}(\mathcal{R}) = \{(t, s) \mid t \in \mathcal{T}_\Sigma, t \rightarrow_{R \cup \Delta, B}^* s\}$.*

Let us also denote by $\text{gnf}(\mathcal{R}) (\subseteq \text{gred}(\mathcal{R}))$ the semantics of ground reducts in normal form, and by $(t, s) \in \text{gnf}(E)$ the fact that s is the canonical form of t w.r.t. the equational theory E .

The theoretical result for the transformation system based on the elementary rules introduced so far (definition introduction, definition elimination, unfolding, folding, and abstraction) are given in [8]. The main result is strong correctness of a transformation sequence, i.e., the semantics of the ground reducts $\text{gred}(_)$ is preserved modulo the equational theory, as stated by Theorem 1.

3.4. Program Semantics and Correctness of the transformation system

Theorem 1 *Let $(\mathcal{R}_0, \dots, \mathcal{R}_k)$, $k > 0$, be a transformation sequence. Then, $\mathbf{gnf}(E_0) =_B \mathbf{gnf}(E_k)$, and for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t, s) \in \mathbf{gred}(\mathcal{R}_0)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathbf{gred}(\mathcal{R}_k)$, $(s, s_2) \in \mathbf{gred}(\mathcal{R}_0)$ and $s_1 =_{E_0} s_2$. Viceversa, for all $t \in \mathcal{T}_{\Sigma_0}$, if $(t, s) \in \mathbf{gred}(\mathcal{R}_k)$ then there exist s_1, s_2 such that $(t, s_1) \in \mathbf{gred}(\mathcal{R}_0)$, $(s, s_2) \in \mathbf{gred}(\mathcal{R}_k)$ and $s_1 =_{E_0} s_2$.*

Chapter 4

Rewrite on Code Carrying Theory

This chapter recall the framework for Code Carrying Theory [18, 19] and show how it can be used to achieve secure delivery of code, explaining how it is designed and how it works. Then we discuss the necessary steps required to combine it with framework of the Code Syntesis described in Chapter 3.

The chapter also introduce the theoretical concept of Certificate as a set of descriptions of transformation operations, and discusses its application to a source theory in order to get an equivalent one with improved performance. We provide some examples of transformation sequences that make evident to the reader the structure and the robustness of the system.

4.1 Background on CCT

Code-Carrying Theory (CCT) [19] is a framework to secure delivery of code. With CCT, instead of transmitting code explicitly like in Proof-Carrying-Code, only assertions and proofs are transmitted to the consumer. If the proof-checking succeeds, the final code is then obtained by applying a simple tool to the resulting theory. CCT can be seen as a means of protecting software from malicious manipulations or eventual corruption of code that

arrives from an untrusted source or through an insecure network, and can be viewed as a variation on the traditional proof-based program synthesis approach. Figure 4.1 illustrates with a diagram the basic architecture of the system.

We can identify two actors in the process: the *Code Consumer* and the *Code Producer*. The Code Consumer defines the requirements, and the Code Producer processes them and provides the Code Consumer with a set of assertions and proofs that satisfy the requirements and improve the system in a secure way. The Code Consumer then can apply a code extractor to the set of assertions and proofs to obtain the executable code, but only if the associated proofs actually do prove the theorems.

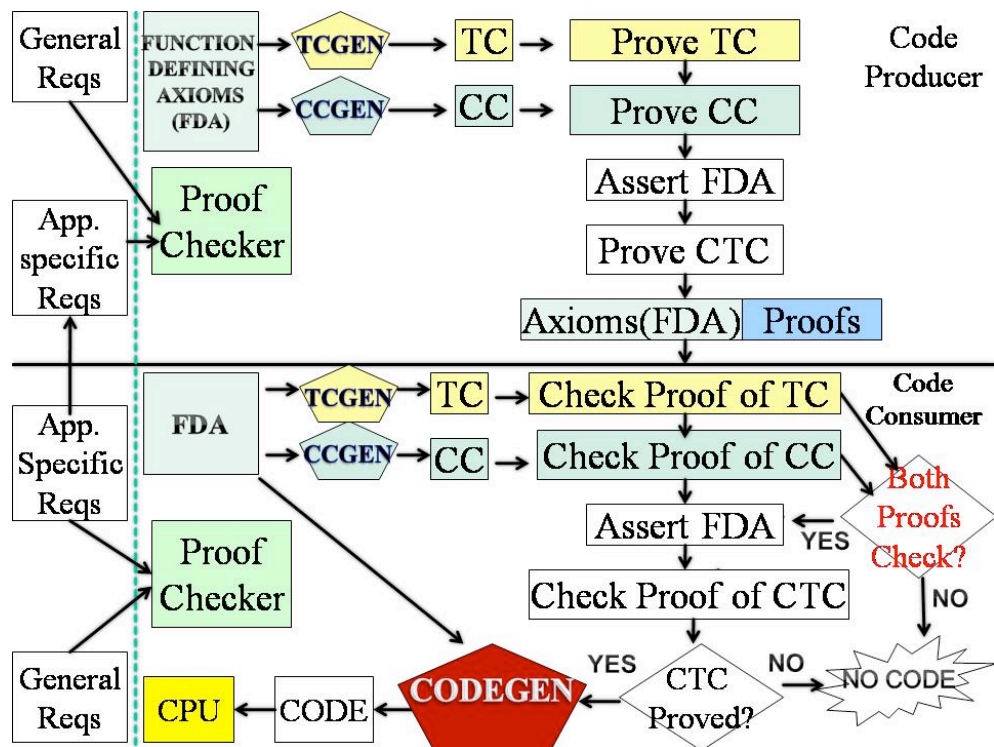


Figure 4.1: Code Carrying Theory Diagram

4.1.1 Code-Carrying Theory Steps

In this section we illustrate all the steps that are necessary to perform a secure transfer of code. At each step we specify who performs the step.

1. ***Code Consumer: Defining Requirements***

The code consumer is responsible for the formulation of requirements. Requirements contain declarations and axiomatic definitions of some common functions. Initially, the code producers do not have these types of requirements and receive them from the consumers later in the process; after that, both consumers and producers share the same set of general requirements.

2. ***Code Producer: Defining a New Function***

Once the code producer receives the specifications of a function requested by the consumer, the code producer wants to send to the code consumer an efficient implementation of the specified function and a proof that it satisfies the required specification. Rather than sending the efficient function as actual code, the producer will send only definitions and proofs in the form of a suitable set of axioms.

3. ***Code Producer: Proving Termination***

The consumer cannot accept and assert arbitrary axioms and propositions. It is the producer's responsibility to show that the new definitions do not introduce any inconsistency into the assumption base. This can be done by proving that the new function-defining axioms satisfy a definitional principle, that takes the form of a requirement that the equations defining the function are total (the function terminates for all inputs). Proving termination is done in TCGEN in Figure 4.1 in an approach that is similar to defining a well-founded ordering onto the equations.

4. ***Code Producer: Proving The Additional Consistency condition***

Similar to Step 3, CCGEN takes a list of function-defining axioms and produces a predicate called consistency condition (CC). The additional consistency condition expresses that it is possible to define a function that satisfies the conditions given by the axioms.

5. ***Code Producer: Proving Correctness***

In this Step, the producer attempts to construct a proof of each of the required correctness conditions by using axioms. Once all the proofs are ready, the producer can send them to the consumer along with the function-defining axioms and proofs of the termination and additional consistency condition. Thus the producer does not send actual code but the theory which carries it.

6. ***Code Consumer: Termination Checking***

After receiving the theory and proofs for a new function, the consumer runs TCGEN to obtain the termination condition (TC). The termination proof sent by the producer must be checked to analyze if it has been corrupted during the transmission or modified by hackers, in which case the consumer will reject the theory immediately, and do not generate the code.

7. ***Code Consumer: Additional Consistency Checking***

Similarly to Step 6, the consumer runs CCGEN to generate the consistency condition (CC) and check it.

8. ***Code Consumer: Checking Correctness***

If the consumer reaches this point, it is safe to assert the new function-defining axioms into the system and check the proofs provided by the producer. If each such proof is accepted, the consumer can access the

last step and generate the code

9. *Code Consumer: Code Extraction*

Now the consumer is sure of the correctness, and the last step can be done. He runs CODEGEN, which extracts the function-defining axioms and generates the final optimized code.

4.2 Program Synthesis and CCT

We have shown that in CCT, only a handy certificate is transmitted in the form of a theory (a set of axioms and theorems) together with a set of proofs of the theorems. No code needs to be explicitly transmitted, hence this is a quasi-natural extension of the framework presented in Chapter 3.

Now we embed the proposed system of Fold/Unfold transformations with CCT methodology, which greatly reduces the operations done by the Code Producer and the Code Consumer. Assuming the Code Consumer provides the requirements in the form of a rewrite theory, the Code Producer can (semi-) automatically obtain an efficient implementation of the specified functions by applying a sequence of transformation rules (*the Certificate*). The Code Producer can then transmit the Certificate, as a compact representation of the transformations sequence, to the Code Consumer who applies it with no need to construct any other correctness proof.

4.2.1 Certificate

Roughly speaking, the Certificate is a representation of the transformations sequences consisting of a sequence of the operations presented in Section 3.3. We provide a formal definition as follows:

Definition 3 (Transformation rule description) *Assuming that rules and equations are referenced by an identification label, a valid transformation rule*

description d is a term of the following forms:

- *Definition Introduction Description:*
Intro(Operator Declaration, Equation Set)
Intro(Operator Declaration, Rule Set)
- *Elimination Description:*
Elim(List of function symbols)
- *Unfolding Description:*
Unfold(Unfolded equation id, Unfold position)
Unfold(Unfolded rule id, Unfold position)
- *Folding Description:*
Fold(Folding equation id, Folded equation id, Fold position)
Fold(Folding rule id, Folded rule id, Fold position)

Definition 4 (Certificate) *Let $(\mathcal{R}_0, \dots, \mathcal{R}_k)$, $k > 0$, be a transformation sequence. The certificate associated with the transformation sequence $(\mathcal{R}_0, \dots, \mathcal{R}_k)$ is the ordered list of transformation rule descriptions $\mathcal{C} \equiv (d_1, \dots, d_k)$ associated with the transformation rules r_1, \dots, r_k s.t. $\forall i \in \{1, \dots, k\}$, r_i is the transformation rule applied to \mathcal{R}_{i-1} in order to obtain \mathcal{R}_i .*

By applying the certificate provided by the Code Producer to the initial inefficient theory (\mathcal{R}_0) , the Code Consumer obtains a new theory semantically equivalent to the first one but with improved performances (\mathcal{R}_k) .

4.2.2 Steps on new framework

With the considered Certificates, so we can revisit the steps defined by the CCT methodology presented in Section 4.1.1 in order to work together in the new framework.

The refined methodology consists only of 3 steps, which are illustrated in Figure 4.2, and summarized below.

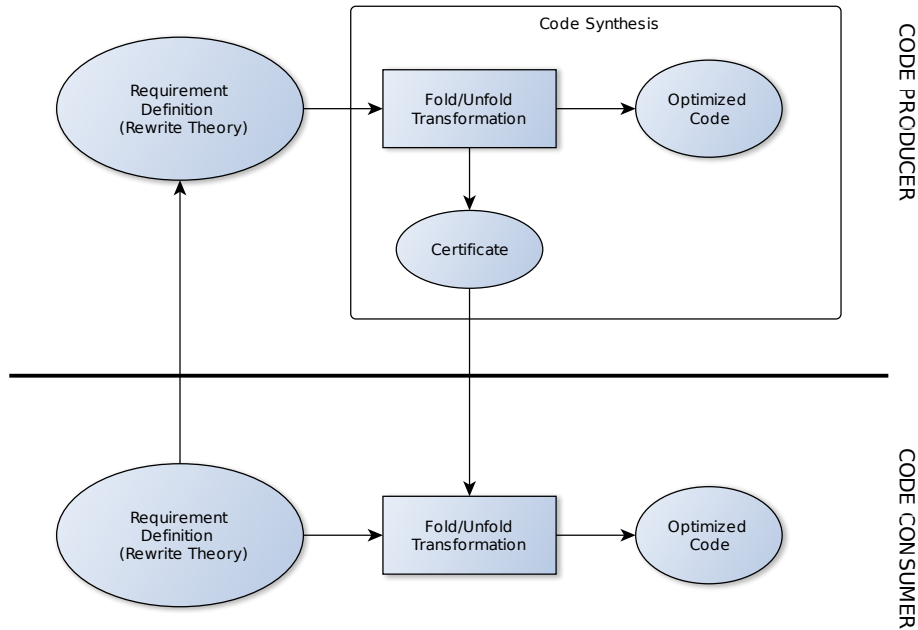


Figure 4.2: Rewrite CCT Diagram

1. *Code Consumer*: Defining Requirements

This step is similar to Step 1 of the basic CCT framework: the Code Consumer provides the requirements to the code producer in the form of a *rewrite theory*. The rewrite theory can be written in Maude, the high-level specification language that implements rewriting logic.

2. *Code Producer*: Defining New Functions

This step resumes the steps [2, ..., 5]. The Code Producer uses the fold/unfold-based transformation system presented in Section 3.3 in order to obtain an efficient implementation of the specified functions. Subsequently, the producer will send only a Certificate \mathcal{C} defined in Definition 4 to be used by the Code Consumer to derive the program.

3. *Code Consumer*: Code Extraction

Once the Certificate is received, the code consumer can apply the transformation sequence, described in the Certificate, to the initial theory, and the final program is automatically obtained. The strong correctness

of the transformation system ensures that the obtained program is correct w.r.t. the initial Consumer specifications., so the Code Consumer does not need to check extra proofs provided by the Code Producer. This covers Steps [6, . . . , 9] of the basic methodology.

4.2.3 Case Study: a tail recursive function

Let us provide an example that explains the overall mechanism described above.

Assume the Code Consumer needs a function for computing the sum of the natural numbers in a list. The consumer specification is a rewrite theory which consists of the equational theory expressed by the following set of rules.

```
op sum-list : NatList -> Nat .
R1 rl sum-list(nil) => 0 .
R2 rl sum-list(x xs) => x + sum-list(xs) .
```

The above rules defining the `sum-list` function can be transformed into a more efficient, tail-recursive structure by using the fold/unfold framework as follows.

Introduce the following new eureka symbol `sum-list-aux`:

```
op sum-list-aux : NatList Nat -> Nat .
R3 rl sum-list-aux(xs,x) => x + sum-list(xs) .
```

Apply the unfold operation over the eureka `R3`, we obtain the following new rules:

```
R4 rl sum-list-aux(nil, x) => x + 0 .
R5 rl sum-list-aux(y ys, x) => x + y + sum-list(ys) .
```

Now, by folding rule $R2$ and $R5$ using the eureka $R3$, we obtain the final tail-recursive program.

```

R1  rl sum-list(nil) ⇒ 0 .
R4  rl sum-list-aux(nil,x) ⇒ x + 0 .
R6  rl sum-list-aux(x xs,y) ⇒ sum-list-aux(xs, x + y) .
R7  rl sum-list(x xs) ⇒ sum-list-aux(xs, x) .

```

The certificate \mathcal{C} is then as follows:

```

(
  Intro((op sum-list-aux : NatList Nat -> Nat.),
        (rl sum-list-aux(xs,x) ⇒ x + sum-list(xs).)),
  Unfold(R3,2),
  Fold(R3,R5,Λ),
  Fold(R3,R2,Λ)
)

```

By applying now the certificate to the initial specification, the code consumer can efficiently obtain the required efficient implementation.

Chapter 5

Security attacks and extension of the framework

In this Chapter, we analyze the security of the transformation framework for rewrite theories applied to CCT described in Chapter 4, and examine a few examples of attacks that witness the main weaknesses of the framework. Then, we discuss how it is possible to prevent the attacks by introducing a suitable procedure for checking the certificates.

5.1 Security

Security is a fundamental aspect of every architecture based on a number of actors that exchange information among them. We have seen in Chapter 4 that the framework for CCT is based on two main actors (Code Consumer and Code Producer) and the data sent between them can be captured by a new malicious actor who could change them, and then cause incorrectness. For instance, the fold/unfold methodology in the CCT context is correct, because the code consumer does not start rewrites from any of the terms of the final theory but only on terms it had previously specified in the original theory, so if a code producer or some intruder in the middle introduces a new function `foo`, the code consumer never starts rewrites with `foo`. If we apply a correct certificate it is impossible to reach terms that carry malicious code

or improper operations.

So if we want identify possible flaws, we need to look for certificates which do not satisfy the conditions of the transformations rules.

In this section, we show a few examples which produce theories that are not correct or complete.

Example 6 Consider the following rewrite theory

$\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, where $X :: s \in \mathcal{V}$ and $\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R with only one sort s , and

$R :$	$R' :$
1. $f(a, a) \rightarrow a$	$f(a, a) \rightarrow a$
2. $f(b, c) \rightarrow d$	$f(b, c) \rightarrow d$
3. $a \rightarrow b$	$a \rightarrow b$
4. $a \rightarrow c$	$a \rightarrow c$
5. $g(X) \rightarrow f(X, X)$	
6.	$g(a) \rightarrow a$

We obtain program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from \mathcal{R} by applying an unfolding step over the rule $g(X) \rightarrow f(X, X) \in R$ which is not *right-linear*, through the narrowing step $f(X, X) \rightsquigarrow_{X/a} a$. Let us consider term $g(a)$. In the original program \mathcal{R} , $g(a)$ can rewrite to the normal form d by the rewrite sequence: $g(a) \rightarrow_5 f(a, a) \rightarrow_3 f(b, a) \rightarrow_4 f(b, c) \rightarrow_2 d$. In the transformed program \mathcal{R}' , it is no longer possible from term $g(a)$ to reach the term d , and, hence, the normal form d is lost. ■

As in the previous example, a Code Producer may send a Certificate with illicit operations which when applied may result in a loss of code functionality. Terms no longer reachable may corrupt a whole system. Particularly they could make the system unstable or potentially attackable, if they had been designed for controlling critical conditions or processing security policies. This is only one possible scenario out of many, and perhaps less dangerous than others, the worst case being those configurations of certificates that lead out of the class of terms that were reached within the initial theory. That

is to say those certificates which cause a loss of correctness, as illustrated in the next example.

When we presented the definition introduction operation, we said that eureka's have to be confluent in order to ensure the correctness of the fold operation. We now discuss this critical point by means of an example.

Example 7 Consider the following rewrite theory

$\mathcal{R} = (\Sigma_{\mathcal{R}}, \emptyset, R)$, obtained from a previous theory \mathcal{R}'' by introducing a fresh symbol m , where $X :: s \in \mathcal{V}$ and $\Sigma_{\mathcal{R}}$ is the signature containing all the symbols of R with only one sort s and

$$\begin{array}{ll}
 R : & R' : \\
 f(a, b) \rightarrow g(a, b) & f(a, b) \rightarrow g(m(a), b) \\
 m(a) \rightarrow a & m(a) \rightarrow a \\
 m(a) \rightarrow b & m(a) \rightarrow b \\
 m(b) \rightarrow a & m(b) \rightarrow a \\
 g(a, X) \rightarrow a & g(a, X) \rightarrow a \\
 g(b, X) \rightarrow b & g(b, X) \rightarrow b
 \end{array}$$

We get program $\mathcal{R}' = (\Sigma_{\mathcal{R}}, \emptyset, R')$ from \mathcal{R} by applying a folding step to the rule $f(a, b) \rightarrow g(a, b)$ using the eureka $m(a) \rightarrow a$. It is easy to see that in \mathcal{R}' we can reduce term $f(a, b)$ to the normal forms a or b , while in \mathcal{R} we can reach only the normal form a . The point is that in \mathcal{R} , term $f(a, b)$ can reduce only to $g(a, b)$ while the fold operation introduces the possibility of rewriting it to $g(b, b)$ cause the eureka defining m is not confluent. This leads to a new solution b , thus missing the correctness. ■

The above example clearly shows that the introduction of non-confluent functions, combined with the fold operation, leads to the achievement of new terms that would otherwise be previously unreachable, and therefore it extends the semantics of ground normal forms considered in the original theory.

5.1.1 Hacking the certificate

Although the code producer should ensure the correctness of the certificate and the framework for fold/unfold transformation ensures the correctness and completeness of the final program, there is no guarantee that the requirements of the transformation rules are met correctly. During delivery, the certificate might be corrupted, or a malicious hacker might change the code. Potential problems can be categorized as security problems (i.e., unauthorized access to data or system resources), safety problems (i.e., illegal operations), or functional incorrectness (i.e., the delivered code fails to satisfy a required relation between its input and output)

If there is no automatic support, it is very easy for the code producer to make a mistake due to the large number of restrictions and preconditions, and it is even easier for an expert malicious hacker to intercept the certificate through an insecure network, modify it and resend to the code consumer. Consider the architecture discussed in Chapter 4 and summarized in Figure 4.2. In Figure 5.1 illustrate a possible attacking scenario, that we comment in the following example of a possible cracking of a certificate.

Example 8 We now reuse the example discussed in Section 4.2.3 and we show what happens when changing the Certificate with functions which do not respect the preconditions. Remember that the consumer specification is a rewrite theory which is expressed by the following set of rules.

```

      op sum-list : NatList -> Nat .
      R1 rl sum-list(nil) => 0 .
      R2 rl sum-list(x xs) => x + sum-list(xs) .

```

The code producer improves the computational cost of this function and produces the certificate \mathcal{C} :

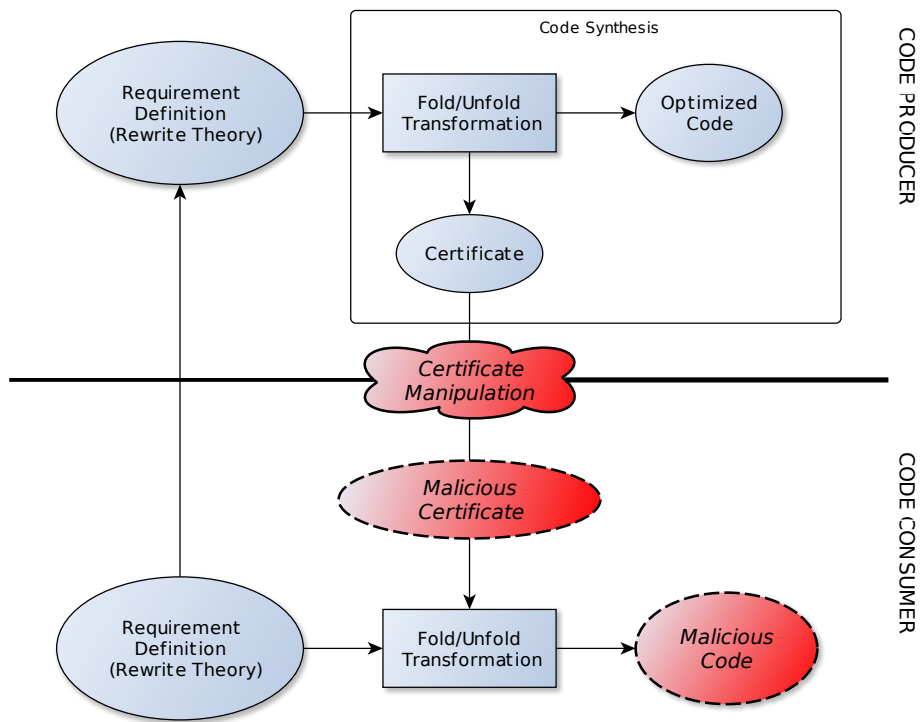


Figure 5.1: Bad Certificate

```

(
  Intro((op sum-list-aux : NatList Nat -> Nat.),
        (rl sum-list-aux(xs, x) ⇒ x + sum-list(xs) .)),
  Unfold(R3, 2),
  Fold(R3, R5, Λ),
  Fold(R3, R2, Λ)
)

```

Now suppose that a malicious hacker changes the \mathcal{C} into \mathcal{C}' as follows:

```

(
  Intro((op sum-list-aux : NatList Nat -> Nat.),
        (rl sum-list-aux(xs, x) ⇒ x + sum-list(xs) .)),
  Intro((op sum-list-aux : NatList Nat -> Nat.),
        (rl sum-list-aux(xs, x) ⇒ (x + sum-list(xs)) + x .)),
  Unfold(R3, 2),
  Fold(R3, R5, Λ),
  Fold(R3, R2, Λ)
)

```

Obviously \mathcal{C}' is not longer a valid certificate, but the code consumer does not realize and applies it obtaining as the final program¹the following set of

¹We skip the intermediate steps of application rules descriptions and show only the final result.

rules:

- $R1$ `rl sum-list(nil) ⇒ 0 .`
 $R4$ `rl sum-list-aux(xs,x) ⇒ (x + sum-list(xs)) + x .`
 $R6$ `rl sum-list-aux(x xs,y) ⇒ y + x + sum-list(xs) .`
 $R7$ `rl sum-list-aux(nil,x) ⇒ x + 0 .`
 $R8$ `rl sum-list(x xs) ⇒ sum-list-aux(xs, x) .`

Now consider the following rewrite sequence as one of the possible computations of the function `sum-list` fed with the list `(1, (1, nil))`:

$$\begin{aligned}
 & \underline{\text{sum-list}((1, (1,\text{nil})))} \rightarrow_{R8} \\
 & \underline{\text{sum-list-aux}((1,\text{nil}), 1)} \rightarrow_{R4} \\
 & (1 + \underline{\text{sum-list}((1,\text{nil}))}) + 1 \rightarrow_{R8} \\
 & (1 + \underline{\text{sum-list-aux}(\text{nil}, 1)}) + 1 \rightarrow_{R4} \\
 & (1 + (1 + \underline{\text{sum-list}(\text{nil})}) + 1) + 1 \rightarrow_{R1} \\
 & (1 + (1 + 0) + 1) + 1 \rightarrow^* 4
 \end{aligned}$$

The result for `sure` is not what one would expect. Actually it is very easy to think of a certificate that makes the system do everything you want. For example if we delete the rule $R6$, the system becomes deterministic and every time the code consumer uses the `sum-list` rule it computes the function $\lambda x.2 * \text{sumlist}(x)$ where `sumlist` is the original function which sums the elements of a list. ■

5.2 Checking the Certificate

In the examples of the previous section we have demonstrated that we cannot apply a certificate regardless of its contents. We need to check that all the operation descriptions to be carried over to the system are lawful (all the

preconditions are respected) and all operations are done in the correct order, because the permutation of the operation descriptions of transformation are not commutative.

In Figure 5.2, we present the extended architecture. Our solution to the problem introduces a procedure to check the certificate which, in case of failure it notifies the code consumer of the poor quality of the certificate, preventing the generation of malicious programs and/or incorrect functions.

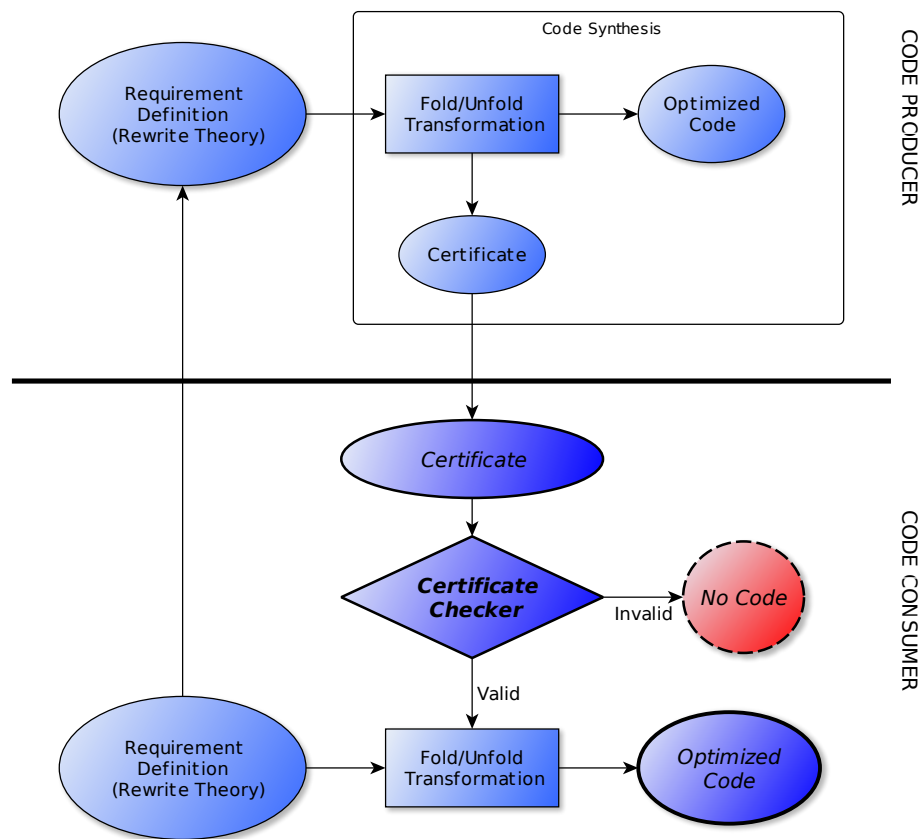


Figure 5.2: Certificate Checking

5.2.1 Preconditions

The basic check is a purely syntactic: we provide a parser for looking that all the functions (*RuleDescription*) defined in Section 4.2.1 which compose a

5.2. Checking the Certificate

certificate respect the following *BNF* grammar.

$$\begin{aligned}
\langle Certificate \rangle &::= (\langle RuleList \rangle) \\
\langle RuleList \rangle &::= \langle RuleDescription \rangle \mid \langle RuleDescription \rangle, \langle RuleList \rangle \\
\langle RuleDescription \rangle &::= \text{introEq}(\langle OpDecl \rangle, \langle EquationSet \rangle) \\
&\mid \text{introRl}(\langle OpDecl \rangle, \langle RuleSet \rangle) \\
&\mid \text{elim}(\langle QidList \rangle) \\
&\mid \text{fold}(\langle Nat \rangle, \langle Nat \rangle, \langle NatList \rangle) \\
&\mid \text{unfold}(\langle Nat \rangle, \langle NatList \rangle) \\
\langle OpDecl \rangle &::= (\text{op } \langle Qid \rangle : \langle TypeList \rangle \rightarrow \langle Type \rangle [\langle AttrSet \rangle] .) \\
\langle EquationSet \rangle &::= \text{none} \mid \langle Equation \rangle \langle EquationSet \rangle \\
\langle Equation \rangle &::= \text{eq } \langle Term \rangle = \langle Term \rangle [\langle AttrSet \rangle] . \\
&\mid \text{ceq } \langle Term \rangle = \langle Term \rangle \text{ if } \langle EqCondition \rangle [\langle AttrSet \rangle] . \\
\langle RuleSet \rangle &::= \text{none} \mid \langle Rule \rangle \langle RuleSet \rangle \\
\langle Rule \rangle &::= \text{rl } \langle Term \rangle \Rightarrow \langle Term \rangle [\langle AttrSet \rangle] . \\
&\mid \text{crl } \langle Term \rangle \Rightarrow \langle Term \rangle \text{ if } \langle Condition \rangle [\langle AttrSet \rangle] . \\
\langle EqCondition \rangle &::= \text{nil} \\
&\mid \langle Term \rangle = \langle Term \rangle \\
&\mid \langle Term \rangle : \langle Term \rangle \\
&\mid \langle Term \rangle := \langle Term \rangle \\
&\mid \langle EqCondition \rangle /\ \langle EqCondition \rangle \\
\langle Condition \rangle &::= \langle EqCondition \rangle \\
&\mid \langle Term \rangle \Rightarrow \langle Term \rangle
\end{aligned}$$

The type of above variable *RuleDescription* is a String provided by Maude and all nonterminals not defined here such as *Term*, *Nat*,... *etc.* are common types of Maude. For a full reference we refer the META-MODULE in the prelude.maude file of the Maude distribution or the Maude manual.

After the syntactic check, we look for precondition violations over the rule descriptions described below:

- Definition Introduction Description check:

The introduced equation (*resp.* rule) is a fresh function symbol which does not occur in the previous transformation sequence, and all sorts in the signature are sorts declared in the original theory provided by the code consumer.

The equations/rules are non-erasing.

Every defined function symbol occurring in right-hand side of a rule belongs to the initial theory.

The set of new equations (*resp.* rules) are left-linear, sufficiently complete² and non-overlapping.

Finally we check also right-linearity of the introduced rules.

- Elimination Description check:

Only symbols not in the original theory can be deleted.

After an elimination, only other elimination rules can be applied. Let x, y be a rule description, where $x \in \{ \textit{Definition Introduction Description}, \textit{Folding Description}, \textit{Unfolding Description} \}$ and $y \in \{ \textit{Elimination Description} \}$; then we accept now only certificates of the form $(x_0, \dots, x_i, y_{i+1}, \dots, y_j)$ or (x_0, \dots, x_i) with $i, j \geq 0$.

- Unfolding Description check:

Due to the use of narrowing described in Section 3.2 by the unfold operation, all preconditions of narrowing must be checked. However, we do this, after the creation of the initial rewrite theory, provided by code consumer.

- Folding Description check:

We remember the signature of the description $\text{folding}(F', F, \text{Pos})$

²We use an external tool [1]

5.2. Checking the Certificate

were F is the folded equation, and F' is the folding equation. The following conditions must be checked: F is not a eureka, and F' is a eureka.

The sort of the function (*resp.* rule) and the term of its application must be preserved.

If all checks are ok, the code consumer is granted that the certificate has it been not manipulated by anyone and none of the attacks discussed in Section 5.1.1 have been done. Therefore he can applies the transformations descriptions to the initial inefficient theory in order to obtain the final optimized program.

Chapter 6

Pyconnect

In this chapter, we provide the motivation that led us to implement Pyconnect. The reader will learn the functionalities, utility and the structure of the program.

In addition we explain in detail how to configure it, and how to extend the source code and its classes in order to adapt at to any ad-hoc situations.

6.1 Description and Motivation

Suppose you want to use a particular software to do a certain operation. Now assume that part of this result should be used by a consumer software. On Unix systems, the concept is well known and is widely used in a wide range of applications: we are obviously talking about the PIPE [13, 17]. If these software systems are developed in such a way as to fit with this concept, the solution is very simple: just call the execution of the two software systems by connecting them in a pipe which serially passes the output of the first one as the input for second one. In this way you can connect in a chain a potentially unlimited number of programs. What happens however, if these software systems are not written according to this paradigm, or the result of the second program must be reused by the first one that at the same time needs to maintain a state of consistency with its own data?

Obviously the solution of the PIPE is not 'appropriate.

Pyconnect, developed in Python a programming language that allows you to work very quickly and integrate your systems more effectively with support for the object oriented paradigm [3, 2]. In our case we need to integrate two environments, in particular Maude [11] that runs the framework for the transformation of programs, and an extended version of Maude 2.3 with Ceta tree automata that is necessary for testing the sufficient completeness [1], as due to incompatibility of versions, we are not able to integrate them directly within the maude language.

Pyconnect is essentially an interactive pipe, which allows us, through a single interface, to transmit data from one or more sources to one or more destinations. Also it allows you to decide which data should be sent on time. Thanks to its modular design, it can be easily extended for ad-hoc solutions. With Pyconnect, we can integrate the two otherwise incompatible environments, and make invisible to the end user the process of transmitting data for the verification of the certificates described in Chapter 5.

6.2 Configuration

Pyconnect manages communication of data, and it creates virtual channels that cannot be created at run-time, but must be declared in the configuration file, so that Pyconnect can create them at startup and manage at runtime. Pyconnect creates FIFOS [13] or named pipes which will be attached to the process. The names of those FIFOS must be specified in the configuration file and are part of the property of the channels created by Pyconnect . The configuration file "pyconnect.config" is situated in the same folder of Pyconnect. We now see in detail its structure and its functionality..

All lines that begin with the character `#` are discarded because they are considered comments by Pyconnect. We have a group of four basic options that help to define the settings for a single channel:

- **NAME:** This variable is used by Pyconnect to recognize the name of the channel and to select it, for routing the data to and from it. The name of the channel must be unique; otherwise all channels with the same name in the environment of Pyconnect intercommunication are excluded.
- **PROC:** This variable indicates the position in the file system of the process that must be started.
- **IN:** This variable indicates the absolute path of the FIFO from which Pyconnect will read. To avoid confusion between those who will read or write, the names refer to Pyconnect, so assuming you create a channel with properties IN, the process attached to that channel must be written on this fifo.
- **OUT:** This variable indicates the absolute path of the fifo on which Pyconnect will write the output data.

To add a new channel it suffices to add into the configuration file a new group of the four variables listed above and their values. The lack of one of the variables in the introduction of a new channel will invalidate the configuration file.

Finally, the variable **DEFAULT** indicate to Pyconnect which of the channels will be selected for communication by default.

6.2.1 Configuration example

Here we can see a typical example of the configuration file, where there are two channels of communication with the first selected one as the default.

```
#simple configuration for Pyconnect

DEFAULT = "maude"

# maude channel
# runs the framework
NAME = "maude"
# #the name of the proc not used
PROC = "/usr/local/bin/maude"
# uses a script to spawn maude like this
# # maude < /tmp/readmeMaude > /tmp/writemeMaude
# #the named pipe used for writing
IN = "/tmp/wirtemeMaude"
# #the named pipe used for reading
OUT = "/tmp/readmeMaude"

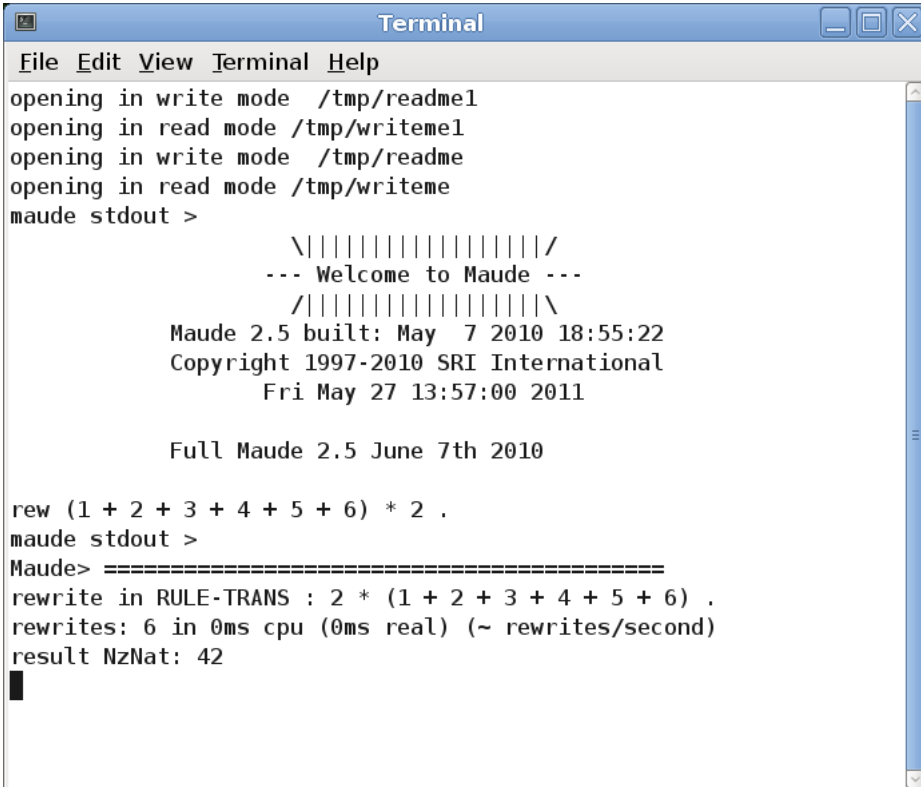
# maude-ceta channel
# runs the sufficient completeness checker
NAME = mceta"
PROC = "/usr/local/bin/maude-ceta"
# uses a script to spawn maude like this
# # maude-ceta < /tmp/readmeMC > /tmp/writemeMC
# #the named pipe used for writing
IN = "/tmp/wirtemeMC"
# #the named pipe used for reading
OUT = "/tmp/readmeMC"
```

6.3 Functions

Once your setup is done, Pyconnect is ready to run. The startup FIFOs specified in the configuration file are created and Pyconnect waits for the connection of the process to them. When all processes are ready, Pyconnect shows you the shell ready for the communication with the channel specified by default.

At this point, the user can enter input commands as if he were in the process. The commands will be passed directly to the selected process and the result will be shown on the screen.

In Figure 6.1, you can see a session of Pyconnect with a channel of Maude.

A terminal window titled "Terminal" with a menu bar containing "File", "Edit", "View", "Terminal", and "Help". The terminal output shows the following sequence of commands and responses:

```
opening in write mode /tmp/readme1
opening in read mode /tmp/writeme1
opening in write mode /tmp/readme
opening in read mode /tmp/writeme
maude stdout >
      \|||||/
      --- Welcome to Maude ---
      /|||||/
Maude 2.5 built: May 7 2010 18:55:22
Copyright 1997-2010 SRI International
Fri May 27 13:57:00 2011

Full Maude 2.5 June 7th 2010

rew (1 + 2 + 3 + 4 + 5 + 6) * 2 .
maude stdout >
Maude> =====
rewrite in RULE-TRANS : 2 * (1 + 2 + 3 + 4 + 5 + 6) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 42
```

Figure 6.1: Pyconnect session

To avoid that commands and controls of the processes overlap, Pyconnect uses a special escape sequence

```
!@#$
```

This sequence is interpreted as the beginning of a command of Pyconnect and must be inserted every time you want to change a channel or modify a property of Pyconnect.

6.3.1 Channels and sending groups

To understand how Pyconnect works and how to use it in an efficient way, we need to understand the concepts of channel and channel group.

Pyconnect is an interactive pipe, but cannot select more than one input/output channels at the same time. Nevertheless this does not affect the ability to send data to other channels: the channel currently selected will have priority over others in that it will write the input of Pyconnect and all the data sent by the user or other connected processes will be sent to it.

The other processes in those channels that are not selected will be in a state of running. Pyconnect does not interfere with their status, but only sends and receives data to be processed, so attention must be paid to synchronization problems among them.

In Figure 6.2 you can see a schematic and detailed representation of the connections.

The important thing to understand is that Pyconnect actually makes no difference between the user and a process. Also the processes can send commands to Pyconnect to change its state.

It might be a good idea to organize a channel/process that supports the other processes, directs the control of the main execution and manages the flow of data when you need it. The communication primitives of Pyconnect are sufficient to direct traffic to a parent process.

Pyconnect organizes channels in a group called sending group, and if the

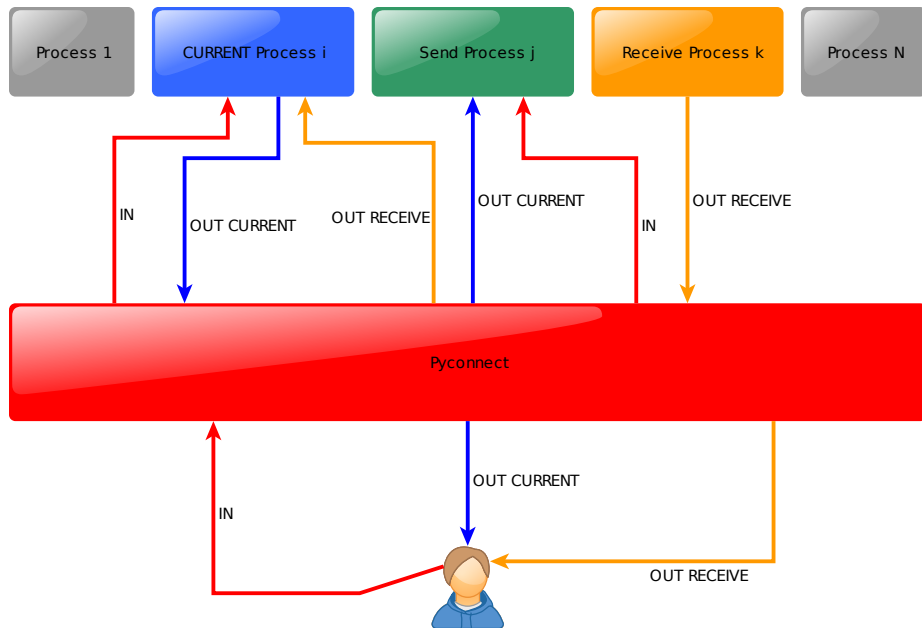


Figure 6.2: Pyconnect Connection Diagram

channel is in this group, then it receives the data sent by Pyconnect. The prompt is always the list of channels that you are connected to and to which the data will be sent.

6.3.2 Pyconnect commands

Let us summarize the commands of Pyconnect with their functionality.

- `!@#$ help` : Prints help screen with the available commands.
- `!@#$ ls` : Prints on screen a list of names of the channels configured by the system.

- `!@#$ select NAME`: Selects a channel for communication. All information will be sent to that channel. NAME refers to the name of the preset channel in the configuration file in the list (and available through the `ls` command). When you select a channel, its output will be sent to Pyconnect and will be sent in accordance with its status.
- `!@#$ send NAME` : adds channel to channel send group. After this operation, the channel will receive an exact copy of the data sent to the selected communication channel.
- `!@#$ unsend NAME` : opposite to the send command, clears the channel NAME from the channel send group.
- `!@#$ receive NAME` : the currently selected channel receives data from channel.

The input of an invalid command or a valid command with an invalid argument, does not change the system status.

6.4 Extending the software

In this section, we discuss the main classes that make up Pyconnect. Pyconnect is written in Python because it is quite versatile and easy to extend. It is organized into a few classes that deal with all the work of routing data. In Figure 6.3, you can see the class diagram. The main class is `RWproc`, which calls the configuration file and initializes the fifos. It is the core of pyconnect, provides methods for high-level reading and writing in a channel, maintains the status and deals with parsing the command line, exchange operations and channel management.

The class `WriterGroup` handles groups of channels and duplicates the input to and from these outwards. However, the important class that performs all the work of timing and data buffering is `IStream` class. This class contains the methods necessary to perform read and write operations to the stream and `parseOut` `parseIn` methods. They are invoked respectively by `read` and `write`, and take a string as input value and return a string as well. In the basic version of `Pyconnect`, these two methods do nothing but return the same string. However, if you extend `Pyconnect` for ad-hoc situations you can redefine these methods or extend the `iStream` class by writing an ad-hoc parser for your own needs. We should point out the abundance of tools for the Python language, including instruments that support regular expressions or others a bit more sophisticated such as `bison`, that can help you to develop your own solution.

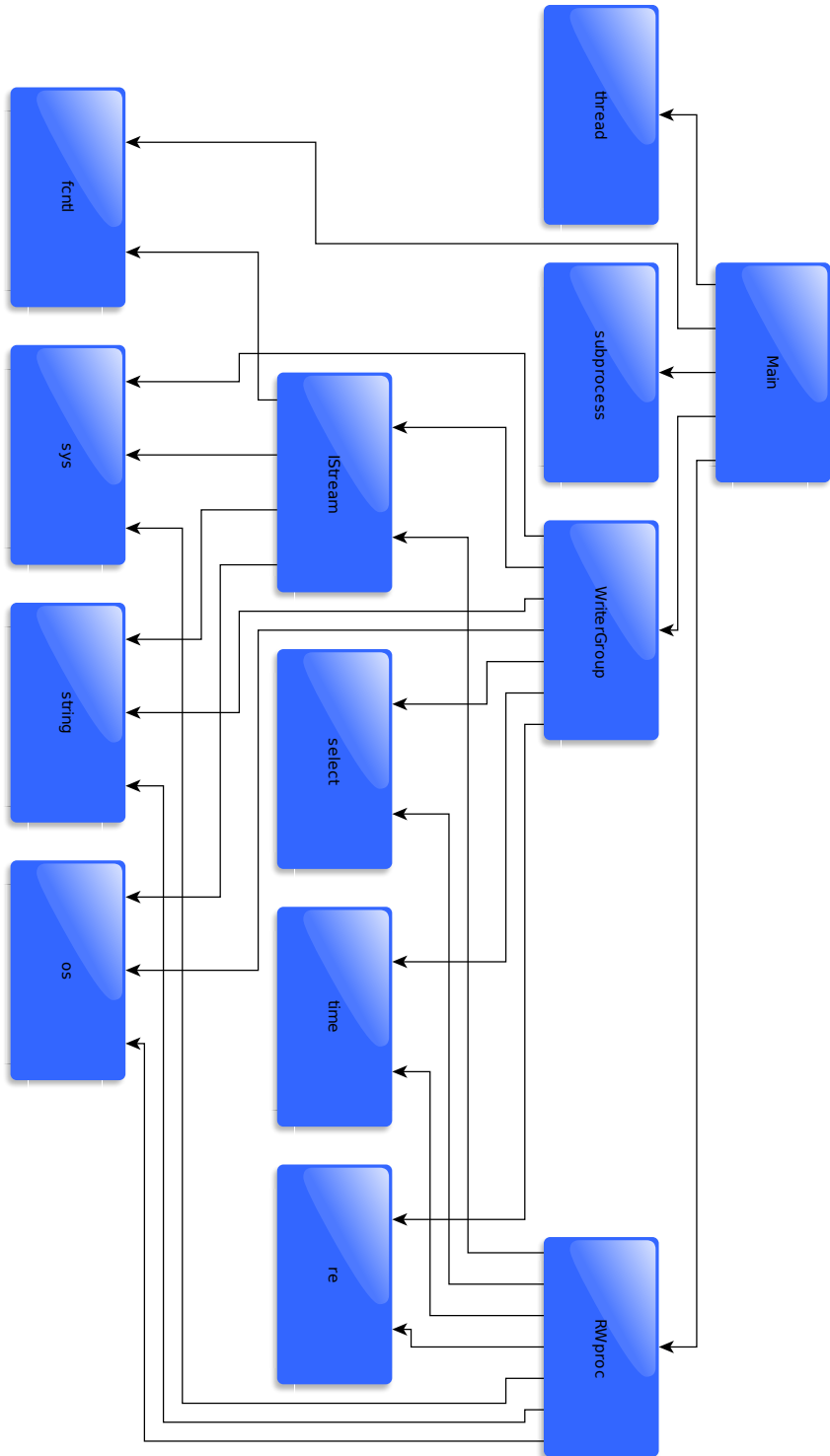


Figure 6.3: Pyconnect Classes Diagram

Chapter 7

Putting all together

In this chapter, we present the resulting framework extended with certificate checking and describe in detail how it is composed and the main features of the user interface.

7.1 Implementation

We implemented in a prototypical system, the transformation framework extended with the infrastructure for certificate checking presented in Chapter 5, which consists of a suit of tools.

The ground components are:

- MetaMaudeCode (MMC), written in Maude 2.5.
- Sufficient completeness checker (SCC), which uses a patched version of Maude 2.3 with ceta tree automata.
- Pyconnect, which support the communication between MetaMaudeCode and the sufficient completeness checker.
- Some scripts in bash that start the frameworks above.

In Figure 7.1, we show the resulting framework architecture. The sufficient completeness checker is a tool written in Maude. Sufficient

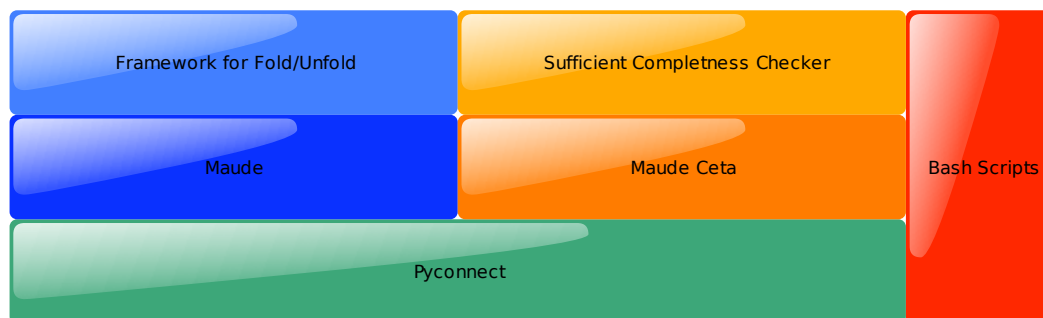


Figure 7.1: Architecture of the framework

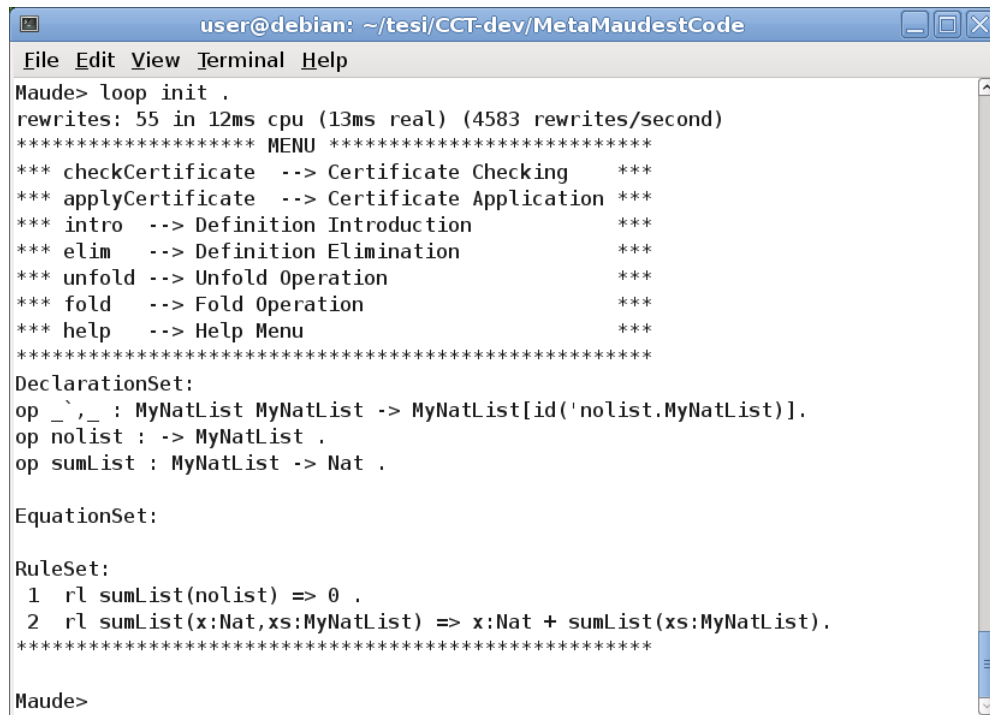
completeness is the property that every operation in a specification is defined on all valid inputs. It is an important property both for developers of specifications (in our case the Code Producer), and the users of these specifications (Code Consumers), to check that they have not missed a case in defined operations. The tree automata SCC [14] version requires an extended version of Maude 2.3, and pre-built binaries for GNU/Linux and x86 processors which are provided by [1].

As explained in Chapter 6, we have implemented Pyconnect due to the incompatibility of the versions of MetaMaudeCode and the Sufficient Completeness Checker: the MMC cannot run in the ceta patched version 2.3, because MMC requires the version 2.5 of Maude, SCC cannot run in a version of Maude without the patch of tree automata and therefore we cannot patch maude 2.5 due to incompatibility and missing codes/patches.

Pyconnect is the main software which manages the connection and the flow of theories, from MMC and SCC, to support us the checking of certificate correctness.

7.1. Implementation

MetaMaudestCode is the core framework for program synthesis and fold/unfold transformation. It is written in Maude 2.5. and provides a function (described below in Section 7.1.1) for testing and applying certificates. A snapshot of the Meta Maudest Code is shown in Figure 7.2 where we have preloaded the example in Section 4.2.3.



```
user@debian: ~/tesi/CCT-dev/MetaMaudestCode
File Edit View Terminal Help
Maude> loop init .
rewrites: 55 in 12ms cpu (13ms real) (4583 rewrites/second)
***** MENU *****
*** checkCertificate --> Certificate Checking ***
*** applyCertificate --> Certificate Application ***
*** intro --> Definition Introduction ***
*** elim --> Definition Elimination ***
*** unfold --> Unfold Operation ***
*** fold --> Fold Operation ***
*** help --> Help Menu ***
*****
DeclarationSet:
op `,_ : MyNatList MyNatList -> MyNatList[id('nolist.MyNatList)].
op nolist : -> MyNatList .
op sumList : MyNatList -> Nat .

EquationSet:

RuleSet:
1 r1 sumList(nolist) => 0 .
2 r2 sumList(x:Nat,xs:MyNatList) => x:Nat + sumList(xs:MyNatList).
*****
Maude>
```

Figure 7.2: MetaMaudestCode

The main scripts which start the Pyconnect, Maude 2.5, Maude 2.3 and the frameworks MMC and SCC, are provided in Bash

7.1.1 Features

The MetaMaudestCode consist of about 2500 lines of code, written in Maude. Basically, our framework allows us to perform the elementary transformation rules and checking of the certificates over a given initial theory. In order to implement the framework MMC, we made use of a useful Maude property

called *reflection*. Rewriting logic is reflective in a precise mathematical way. In a finitely presented universal theory \mathcal{U} we can represent in \mathcal{U} any finitely presented rewrite theory \mathcal{R} as a meta-term $\overline{\mathcal{R}}$, any term t, t' in \mathcal{R} as meta-terms $\overline{t}, \overline{t'}$, and any pair (\mathcal{R}, t) as a meta-term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that we have the following equivalence:

$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$ because \mathcal{U} is representable as a meta-term itself, it is possible to extend the equivalence to $\mathcal{U} \vdash \langle \overline{\mathcal{U}}, \langle \overline{\mathcal{R}}, \overline{t} \rangle \rangle \rightarrow \langle \overline{\mathcal{U}}, \langle \overline{\mathcal{R}}, \overline{t'} \rangle \rangle$.

Thanks to Maude reflection, our framework has been easily implemented by manipulating the meta-term representations of rules and equations. In practice, both transformation rules and certificate checking presented respectively in Sections 3.3 and 5.2, have been implemented as rewrite rules that work and manipulate the meta-term representation of the rewrite theories we want to transform and the transformation rule descriptions that perform it.

The interface allows one to perform single transformation rules over the initial theory (we do not recommend it for Code Consumers but only for testing purposes by Code Producers), checking certificates syntactically, checking the certificates preconditions (we recommend to perform this operation by both, Producer to avoid mistakes/omissions and Consumers to avoid corruption/cracking), and applications of the Certificate.

When the user performs a transformation sequence, the result is shown step by step in the form of intermediate theories. When the user performs a certificate checking, a report is shown over the single steps, and in the case of bad certificates the cause of failure is reported, as well as in specific cases where the certificates do not respect the correctness preconditions.

In Figure 7.3 we present the modules structure used by our framework, which can be useful for future extensions or maintenance of the software.

Chapter 8

Conclusions

In this dissertation, we investigated on some of the most recent and complex directions of research in computer science and we proposed some effective solutions. We extended a novel framework for Code Carrying Theory, which is fed with a methodology based on narrowing to perform fold/unfold program transformation. In order to achieve this goal, we made use of rule-based formalisms, such as rewriting logic, and narrowing to develop our system. The main aspect in which we concentrated our focus is the security and the interoperability of the framework.

The core transformation rules of our CCT framework are folding, unfolding, definition introduction, and definition elimination. The correctness of the program transformation framework guarantees that the transformed program is equivalent to the initial one, and the program synthesis methodology can be effectively applied to CCT and significantly simplify the code producer and code consumer tasks. More precisely, the transformation process is represented as a compact sequence of applied transformation rules, and is delivered as a certificate to the code consumer. The distributed character of the considered systems has led us to the study of security aspects such as the software certification for secure the delivery of code, and we have shown that the code consumer cannot apply a certificate regardless of its contents, due to the possibility of certificate manipulation by a malicious actor that can attack the whole system. To obtain the desired final and improved program,

the code consumer needs to check and apply the certificate to the initial requirements, which requires only modest computational resources. Future work related to the subject presented in this thesis includes experiments with optimizations to speed up the analysis, and automatize the code synthesis process.

We implemented in a prototypical system the transformation framework and extended it with the infrastructure for certificate checking, which implements the complete CCT infrastructure, reducing the number of the steps and the burden of the code producer and the code consumer. So the code consumer which uses our framework can receive, check and apply a certificate to an initial theory in order to obtain the desired program, can detect and refuse bad certificates with a detailed report; and then avoid data corruption or attacks from malicious actors.

We also plan to take advantage of the Pyconnect architecture, which allows us to integrate in our framework, in an easy way, some available Maude formal tools, to verify other relevant program properties such as termination and confluence of the initial equations set of the theory. Moreover, we can integrate an automated theorem prover for the verification of interested consumer properties. Such an extension is subject to future work.

Bibliography

- [1] <http://maude.cs.uiuc.edu/tools/scc/>, 2011.
- [2] *Programming Python, 3rd Edition*. O'Reilly, August 2006.
- [3] <http://python.org>, 2011.
- [4] Bouhoula A., Jouannaud J.P., and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.
- [5] M. Alpuente, M. Baggi, D. Ballis, and M. Falaschi. A fold/unfold framework for rewrite theories extended to cct, 2009. Submitted at: Workshop on Partial Evaluation and Program Manipulation (PEPM '10).
- [6] M. Alpuente, M. Baggi, D. Ballis, and M. Falaschi. Maudeniccate program transformation system. Available at <http://users.dimi.uniud.it/~michele.baggi/cct/>, 2009.
- [7] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [8] M. Baggi. *Rule-based Methodologies for the Specification and Analysis of Complex Computing Systems Rule-based Methodologies for the Specification and Analysis of Complex Computing Systems*. PhD thesis, Universidad Politecnica de Valencia, 2009.
- [9] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Unification and narrowing in maude 2.4. In Ralf Treinen, editor, *Rewriting Techniques and Applications, 20th*

International Conference, RTA 2009, Brasília, Brazil, 2009, Proceedings, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

- [10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.
- [11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [12] M. Fay. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction*, pages 161–167, 1979.
- [13] GNU. *Linux Programmer's Manual*, 2010.
- [14] Hitoshi Ohsaki Joe Hendrix and José Meseguer. Sufficient completeness checking with propositional tree automata. Technical report, University of Illinois, National Institute of Advanced Industrial Science and Technology, and PRESTO, Japan Science and Technology Agency.
- [15] J.W. Klop. *Term Rewriting Systems*. 1989.
- [16] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher Order Symbolic Computation*, 20(1-2):123–160, 2007.
- [17] Simone Piccardi. *GaPiL Guida alla Programmazione in Linux*. luglio 2010.
- [18] A. Vargun. *Code-carrying theory*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 2006. Adviser-D.R., Musser.
- [19] A. Vargun and D.R. Musser. Code-carrying theory. In *ACM symposium on Applied computing*, pages 376–383, New York, NY, USA, 2008. ACM.

- [20] P. Viry. Rewriting: An effective model of concurrency. In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 648–660, London, UK, 1994. Springer-Verlag.

List of Figures

4.1	Code Carrying Theory Diagram	22
4.2	Rewrite CCT Diagram	27
5.1	Bad Certificate	35
5.2	Certificate Checking	38
6.1	Pyconnect session	47
6.2	Pyconnect Connection Diagram	49
6.3	Pyconnect Classes Diagram	52
7.1	Architecture of the framework	54
7.2	MetaMaudestCode	55
7.3	MetaMaudesCode Inclusion Modules Diagram	57